Christian-Albrechts-Universität Kiel
Institut für Informatik
Arbeitsgruppe Softwareentwicklung

Diplomarbeit

Ein 3D-Ansatz zur Visualisierung der Kernauslastung in Multiprozessorsystemen

Björn Konarski (bko@informatik.uni-kiel.de)

15. Juli 2012

Betreut durch: Prof. Dr. Wilhelm Hasselbring
Dipl.-Inf. Jan Waller

Inhaltsverzeichnis

Zusammenfassung

Eine gleichmäßige Verteilung der Last in einem Multiprozessorsystem ist ein Eckpfeiler einer stabilen und schnell reagierenden Software. Um Entwicklern ein Werkzeug in die Hand zu geben, mit dem sie das Verhalten ihrer Software in einem Multiprozessorsystems analysieren und Flaschenhälse beseitigen können, wird in dieser Arbeit ein Ansatz zu Visualisierung der Laufzeit von Threads auf einem logischen Prozessor vorgestellt. Dazu werden zunächst verschiedene Visualisierungstechniken und ein System zur Einordnung und Bewertung vorgestellt. Anschließend werden das Konzept und die Implementierung eines Prototypen zu diesem Ansatz vorgestellt. Abschließend wird der implementierte Prototyp anhand der beschriebenen Kriterien evaluiert und mögliche Verbesserungen vorgeschlagen.

Version

Zuletzt erstellt:
15. Juli 2012
22:13Uhr

Danksagungen

Ich möchte an dieser Stelle allen Danken (mit großem D), die mich während der Erstellung dieser Arbeit unterstützt haben, entweder direkt durch Kritik und Hilfe oder indirekt dadurch, das sie mich die letzten sechs Monate ertragen haben, ohne mich zu erwürgen.

Eidesstattliche Erklärung

Ich versichere, dass ich diese Arbeit selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und alle wörtlich oder sinngemäß übernommenen Textstellen als solche gekennzeichnet habe.

Kiel, den 15. Juli 2012

1 Einleitung

1.1 Motivation

Der Trend in der Chipherstellung hat sich in den letzten Jahren gewandelt. Der Wettlauf um immer höhere Taktraten ist der Entwicklung von Prozessoren mit immer mehr Kernen und immer stärkerer Parallelisierung gewichen [Hruska 2012]. Dadurch ist es für Softwareentwickler notwendig, ihre Anwendungen so zu gestalten, dass sie in der Lage sind, alle zur Verfügung stehenden Kerne optimal auszulasten. Durch diese Optimierung kann in vielen Anwendungen an Geschwindigkeit und Reaktivität gewonnen werden, so dass sich die gesamte Anwendung flüssiger anfühlt. Besonders Anwendungen aus den „klassischen“ Einsatzgebieten für Multiprozessorsysteme, beispielsweise 3D-Grafikanimation oder „Number Crunching“ sehr großer Datenmengen, können davon profitieren.

Um den Trend der „Parallelisierung von Software“ weiter voranzutreiben, ist es zunächst notwendig, die benötigten Werkzeuge zur Verfügung zu stellen. Diese Werkzeuge müssen so gestaltet sein, dass sie nicht nur den reinen Verbrauch an Prozessorzyklen analysieren, sondern auch das Verhalten einzelner Threads wiedergeben. Durch graphische Aufbereitung und Darstellung des Schedulings einzelner Threads können Entwickler feststellen, ob ihre Anwendung ausreichend parallelisiert sind, um einen Vorteil aus mehreren verfügbaren Prozessoren zu ziehen.

Ein weiterer Bereich, in dem eine graphische Darstellung einzelner Threads und des Prozessors, auf dem sie ausgeführt werden, sinnvoll ist, ist die Entwicklung von Schedulingmechanismen selbst. Ein Entwickler bekommt dadurch ein schnelles visuelles Feedback, ob sein Algorithmus die an ihn gestellten Anforderungen erfüllt.

Die vorliegende Arbeit stellt einen Ansatz vor, wie ein solches graphisches Werkzeug zur Analyse der Kernauslastung aussehen kann.

1.2 Ziele der Arbeit

Das erste Ziel dieser Arbeit ist das Erstellen von Anforderungen an ein Werkzeug, das die Kernauslastung eines Multiprozessorsystems darstellen soll. Das Hauptaugenmerk bei der Erstellung der Anforderungen liegt dabei auf der graphischen Darstellung des Werkzeugs sowie seiner Bedienung. Dazu werden bestehende Arbeiten ausgewertet, die sich mit der Bewertung und Einordnung von Visualisierungstechniken beschäftigen. Aus den bestehenden Arbeiten werden Kriterien abgeleitet, anhand derer eine mögliche Visualisierung eingeordnet werden können. Gleichzeitig dienen diese Kriterien als Grundlage für die Anforderungen, die an die Umsetzung einer solchen Visualisierung gestellt werden.

Das zweite, praktische Ziel der Arbeit ist die Erstellung eines konkreten Prototyps auf Basis der im ersten Teil ermittelten Anforderungen. Damit soll Entwicklern ein Werkzeug an die Hand gegeben werden, das sie bei der Analyse ihrer Software unterstützt.

1.3 Gliederung der Arbeit

Der Anfang dieser Arbeit, bestehend aus den Kapiteln 2 und 3, befasst sich mit dem theoretischen Teil, in dem die Grundlagen für die Entwicklung einer Visualisierung geschaffen wurden. Der Mittelteil der Arbeit beschäftigt sich mit der Planung, Ausführung und Bewertung des erstellten Prototypen. Dieser Teil erstreckt sich über die Kapitel 4, 5 und 6. Den Abschluss bilden Kapitel 7, das einen Ausblick auf mögliche Erweiterungen und Folgearbeiten gibt, und 8, das die Arbeit zusammenfasst und abschließt.

2 Einordnung und Bewertung

Das folgende Kapitel beschreibt verschiedene Arbeiten zum Thema Visualisierung. Dabei geht es um zum Einen um die Frage, wie wir eine Visualisierung einordnen können, also ähnliche Formen der Visualisierung zusammenfassen können, zum Anderen um die Bewertung einer Visualisierung. Die hier vorgestellten Arbeiten bieten Ansätze dazu.

Aus den bestehenden Ansätzen zur Einordnung und Bewertung soll eine Liste von Kriterien herausgefiltert werden, die für eine Visualisierung wichtig sind. Diese Liste wird um Kriterien erweitert, die für den Prototypen zusätzlich relevant sind.

Ziel des Kapitels ist es, die Grundlagen zu beschreiben, auf denen die weiteren Designentscheidungen des Prototypen aufbauen. Zunächst werden einige bereits existierende Arbeiten zum Thema Visualisierungen beleuchtet. Anschließend wird eine Übersicht der für die Einordnung und Bewertung relevanten Kriterien gegeben.

2.1 Bestehende Kriterien / Einordnungen

In diesem Abschnitt werden bereits vorhandene Arbeiten von Freitas et al. [2002], Sebrechts et al. [1999], Fuhrmann and Gröller [1998], Chi [2000], Shneiderman [1996] und Tominski [2011] zusammengefasst. Die vorgestellten Arbeiten liefern Kriterien, die entweder der Bewertung oder Einordnung von Visualisierungen dienen.

2.1.1 Ansatz von Freitas et al. [2002]

Freitas et al. [2002] beschreiben einen kurzen, allgemeinen Ansatz zur Evaluierung von Visualisierungen. Dieser Ansatz soll einem Entwickler oder Designer helfen, die Qualität ihrer Visualisierung zu beurteilen und gegebenenfalls zu verbessern. Es werden vier Kriterien für das Bewerten von Visualisierungen vorgestellt: *Vollständigkeit*, *Räumliche Aufteilung*, *Aufschlüsselung der Informationen* und *Zustandsübergänge*. Anhand der Kriterien kann eine Visualisierung auf ihre *Ausdrucksstärke* und ihre *Bedienbarkeit* geprüft werden.

Vollständigkeit Vollständigkeit bezieht sich auf die Vollständigkeit der dargestellten Daten. Eine Visualisierung ist vollständig, wenn sie alle Daten, die ihr zur Verfügung stehen, darstellen kann. Für das Kriterium der Vollständigkeit ist die Art und Weise, wie die die Daten angezeigt werden, nicht von Bedeutung. Mehr dazu siehe Kap. 2.2.1.

Räumliche Aufteilung Unter räumlicher Aufteilung sind Eigenschaften des Layouts zusammengefasst, also wie Objekte zueinander positioniert sind, ob Objekte voneinander verdeckt werden o.ä. Auch der logische Aufbau des Layouts ist hier zusammengefasst, also ob Objekte, die bestimmte Daten darstellen, in Gruppen dargestellt sind. Die Anordnung der Gruppen zueinander sollte einem erkennbaren Muster folgen. Mehr unter Kap. 2.2.3.

Aufschlüsselung der Informationen Unter diesem Punkt sind Eigenschaften wie Farben und die Verwendung bestimmter Symbole zusammengefasst. Dieses Kriterium befasst sich mit der Art und Weise, auf die Daten angezeigt werden. Ein Beispiel wäre die Bedeutung der Größe eines Objektes. Ist das Objekt größer, ist der durch das Objekt repräsentierte Wert höher. Diese Information muss an den Benutzer weitergegeben, also aufgeschlüsselt werden. Siehe auch Kap. 2.2.5.
Zustandsübergänge Das Kriterium Zustandsübergänge ordnet Visualisierungen nach der Zeit, die sie benötigen, um eine Änderung anzuzeigen. Eine Änderung ist ein durch den Benutzer angestoßener Vorgang, der die räumliche Aufteilung oder einen anderen Aspekt der Darstellung verändert.

Diese vier Kriterien werden ergänzt um drei weitere Kriterien zur Bewertung der Bedienmechanismen: *Orientierung und Hilfe*, *Navigation und Abfrage* und *Verkleinerung des Datensatzes*.
Orientierung und Hilfe Orientierung und Hilfe beschreibt die Möglichkeiten des Nutzers, sich innerhalb der Visualisierung zurechtzufinden. Dazu gehören Mechanismen wie Label, die den momentanen Modus eines Texteditors anzeigen („Editieren“ oder „schreibgeschützt“). Innerhalb einer 3D-Landschaft wäre ein entsprechender Mechanismus das Einblenden der Achse oder anderer Orientierungspunkte.
Navigation und Abfrage Dieser Punkt fasst die Mechanismen der Visualisierung zusammen, die es dem Benutzer erlauben, Daten innerhalb eines Datensatzes zu finden. Sind für den Benutzer beispielsweise alle (lokalen) Maxima eines Graphen relevant, könnten diese in der Visualisierung farblich hervorgehoben werden. Auch die Möglichkeit, Stichpunktsuchen zu interessanten Daten anzustellen, wird unter diesem Kriterium zusammengefasst.
Verkleinerung des Datensatzes Die Verkleinerung des Datensatzes beschreibt die Möglichkeit für den Benutzer, die Darstellung des gegebenen Datensatzes auf eine Teilmenge einzugrenzen. Beispielsweise interessiert sich ein Benutzer nur für einen bestimmten Zeitraum innerhalb eines Datensatzes. Denkbar sind auch komplexere Filter, die aus mehreren Bedingungen zusammengesetzt sind. Ein Benutzer könnte so die Darstellung beschränken auf „alle Bücher in dieser Bibliothek, die zwischen 1990 und 1997 verfasst wurden, mehr als 200 Seiten haben und in Deutschland veröffentlicht wurden“. Dies setzt voraus, das die zu filternden Daten (Datum, Seitenzahl...) für jeden Datensatz zur Verfügung stehen.

2.1.2 Unterscheidung und Bewertung von Text-, 2D- und 3D-Ansätzen

Sebrechts et al. [1999] vergleichen Text-, 2D- und 3D-Ansätze von Visualisierungen anhand von Suchaufgaben, die einer Reihe von Testpersonen gestellte wurden. Die Aufgabe bestand darin, aus einer gegebenen Menge an Dokumenten bestimmte Informationen herauszusuchen. Dabei wurde der Fokus auf die Benutzbarkeit des Interfaces gelegt. Es

wurden funktional möglichst ähnliche Interfaces verglichen, um allgemeine Aussagen über Vor- oder Nachteile der entsprechenden Darstellung treffen zu können. Ein Vergleichskriterium sind die Zeiten, die Testpersonen zum Auffinden bestimmter Informationen aus einer großen Menge anderer Informationen benötigen. Es wurden zwei Ergebnisse ermittelt. Ein *quantitatives* Ergebnis, das anhand der von den Testpersonen benötigten Zeit ermittelt wurde, und ein *qualitatives* Ergebnis, das auf den subjektiven Bewertungen der Testpersonen beruht.

Das quantitative Ergebnis der Untersuchung ist, dass die Dimensionalität der Darstellung einen geringeren Einfluss auf die Verwendbarkeit hat, als die Auswahl an Funktionalitäten oder die Aufgabe selbst. Für den hier vorgestellten Prototypen bedeutet dies, das allein die Implementierung in 3D keinen ausreichenden Vorteil darstellt. Daher wurde der Fokus auf die Funktionalität verstärkt. Das qualitative Ergebnis der Studie erfasste folgende Punkte:

Gruppierung von Dokumenten (zu Clustern) Die von den Testpersonen zu durchsuchenden Dokumente wurden anhand bestimmter Dokumenteigenschaften gruppiert. Dies hat sich als vorteilhaft für die Übersichtlichkeit herausgestellt.

Farbkodierung von Eigenschaften Verschiedene Eigenschaften der zu durchsuchenden Dokumente wurden in verschiedenen Farben hervorgehoben. Dies wurde von den Testpersonen sehr positiv aufgenommen und verkürzte die zum Durchsuchen benötigte Zeit stark.

Sicht- und Lesbarkeit Mit steigender Anzahl an Eigenschaften und sinkender Textgröße in 2D- und 3D-Umgebungen nahm die Lesbarkeit der Darstellung deutlich ab. Die Autoren betonen, das Kompromisse zwischen textueller und graphischer Darstellung von Fall zu Fall geschlossen werden müssen.

Reaktionszeit und Komplexität des Systems Eine möglichst geringe Reaktionszeit wurde von den Testpersonen als angenehm empfunden. Große Datenmengen können zwar noch immer zu langen Verarbeitungszeiten führen, jedoch ist die Leistung moderner Rechner inzwischen derart gewachsen, das die Unterschiede zwischen textuellen und 2D- bzw. 3D-Darstellungen heutzutage nicht mehr ins Gewicht fallen. Besonders unerfahrene Testpersonen waren auch durch eine zu große Anzahl ausführbarer Aktionen überfordert.

Räumliche Orientierung Dieser Punkt betrifft vornehmlich die getestete 3D-Darstellung. Einige Testpersonen verloren während der Bewegung im dreidimensionalen Raum die Orientierung. Auch das hinzufügen von Orientierungspunkten schaffte keine Abhilfe.

2.1.3 Einordnung auf Basis eines State Reference Models

In Chi [2000] wird eine Einordnung von Visualisierungstechniken auf Basis eines „Data State Reference Models“ beschrieben. Dabei wird lediglich eine Einordnung vorgestellt, eine Bewertung wird nicht vorgenommen. Das Modell teilt den Vorgang der Visualisierung in vier aufeinander folgende Zustände: *Wert*, *Analytische Abstraktion*, *Visuelle Abstraktion* und *Ansicht*.

Wert Der erste Zustand einer Visualisierung ist der *Wert*. Damit ist die Form gemeint, die die gesammelten Rohdaten haben. Dies können einfache Zahlenlisten oder komplexe Objekte wie eine Menge von Büchern in einer Bibliothek sein.
Analytische Abstraktion In diesem Zustand sind die Daten in einer Form, die eine Analyse erlauben. Den Rohwerten aus dem vorherigen Schritt wurden Metadaten hinzugefügt, die dem Rohwert Bedeutung geben. Den Zahlenlisten aus dem obigen Beispiel wurde zugeordnet, das es sich um Messwerte eine Gasmessgerätes handelt. Jedem Buch in der Bibliothek wurden beispielsweise Titel, Autor und Erscheinungsjahr zugeordnet. Man betrachtet auf dieser Stufe *Informationen*, d.h. die Rohdaten werden nur noch in Zusammenhang mit ihren Metadaten betrachtet.
Visuelle Abstraktion Die visuelle Abstraktion ist die Aufbereitung der vorhandenen Informationen in eine Form, die von einer Visualisierungstechnik dargestellt werden kann. Die Messwerte aus dem Gasmessgerät werden auf eine Zeitachse aufgetragen, die Bücher werden in eine Tabelle mit Spalten für Titel, Autor und Erscheinungsjahr gebracht.
Ansicht Die Ansicht ist tatsächliche, bildhafte Ansicht der aufbereiteten Daten, beispielsweise in Form eines Graphen oder einer Karte. Die Messwerte sind als Punkt in einem Graphen aufgetragen und durch ergänzende Linien verbunden. Die Bücher sind in tabellarischer Form in ein Textdokument eingebunden, eine Zeichenkette repräsentiert Titel, Autor und Erscheinungsjahr.

Zwischen jedem Zustand wird nach der Art der Transformation unterschieden. Eine mögliche Transformation zwischen visueller Abstraktion und Ansicht wäre das umwandeln kommaseparierter Werte in einen zweidimensionalen Graph durch Auftragen der Werte auf den Achsen. Diese Transformationen dienen zum Übergang zwischen den Zuständen. Innerhalb eines Zustands können sogenannte *innere Operationen* auftreten. Innere Operationen verändern die Daten innerhalb des momentanen Zustands, ohne in einen anderen Zustand zu wechseln. Darunter fallen Operationen wie das Filtern (z.B. das Glätten einer Statistik innerhalb der analytischen Abstraktion) oder Sortieren von Werten (z.B. das alphabetische Sortieren einer Tabelle in der visuellen Abstraktion). Die eigentliche Einordnung verschiedener Visualisierungen erfolgt anhand der Unterscheidung der Zustände, der Transformationen und der inneren Operationen.

2.1.4 Einordnung auf der Basis „Task by Data Type“

Die von Shneiderman [1996] vorgestellte „Task by Data Type Taxonomy“ bezeichnet eine Einordnung auf Basis von *Kategorien von Datentypen*, zu denen *Funktionen* definiert sind. Da Ziel ist, dass ein Entwickler zunächst seine Daten in eine bestimmte Kategorie einordnen kann, um später die dazugehörigen Funktionen implementieren zu können. Die Funktionen sollen dem Anwender helfen, Aufgaben zu erledigen, die gewöhnlich für diesen Datentyp anfallen.

Zuerst beschreibt Shneiderman [1996] die Kategorien für Datentypen. Die Kategorisierung wird nach der visuellen Struktur der Daten vorgenommen, also wie ein Datenobjekt dem Anwender präsentiert wird. Jedes Datenobjekt kann eine beliebige Menge von Attributen besitzen.

Es sind folgende Kategorien von Datentypen definiert: *Eindimensional*, *Zweidimensional*, *Dreidimensional*, *Temporal*, *Multidimensional*, *Baum*, *Netzwerk*.

Eindimensional Als Eindimensionale Daten werden lineare Daten wie Quellcode oder Textdokumente bezeichnet. Jede Zeile stellt ein einzelnes Datenobjekt dar, dessen Attribute beispielsweise Zeilennummer oder Fontgröße sein können. Eine mögliche Aufgabe wäre das Auffinden einer bestimmten Zeile.

Zweidimensional Zweidimensionale Datentypen sind Daten, die beispielsweise als Karten oder Layouts vorliegen. Ein Datenobjekt wird dabei durch die Fläche definiert, die es einnimmt. Ein zusätzliches Attribut in einer Karte wäre der Name einer Stadt oder einer Straße. Mögliche Aufgaben sind u.a. das Finden von benachbarten Datenobjekten oder das Finden von Wegen von einem Datenobjekt zu einem Anderen.

Dreidimensional Dreidimensionale Datentypen sind Gegenstände der echten Welt oder ihre Simulation. Jeder dreidimensionale Gegenstand ist ein Datenobjekt. Dreidimensionale Daten verhalten sich ähnlich wie zweidimensionale Daten, durch die zusätzliche Dimension kommen jedoch Aufgaben wie das Bestimmen der Ausrichtung oder das Vermeiden von Verdeckung hinzu.

Temporal Temporale Datentypen sind verwandt mit eindimensionalen Datentypen. Der Unterschied ist dabei, dass ein Datenobjekt nicht definiert ist durch den physischen Platz, den er einnimmt, sondern durch die Zeit, in der er geschieht. Jedes Datenobjekt besitzt einen Startzeitpunkt und einen Endzeitpunkt. Dabei können Datenobjekte einander beliebig überlappen. Ein Beispiel für eine Aufgabe wäre das Auffinden aller parallel geschehenden Datenobjekte.

Multidimensional Multidimensionale Datentypen umfassen komplexe Datenformen wie relationale Datenbanken. Datenobjekte dieses Typs sind Punkte in einem n-dimensionalen Raum. Die tatsächliche Darstellung der Datenobjekte wird daher in zwei- oder dreidimensionale Repräsentationen heruntergebrochen. Eine Beispielaufgabe wäre das Auffinden von Verknüpfungen zwischen zwei Datenobjekten.

Baum Ein Baum ist eine streng hierarchische Struktur, bestehend aus Kanten und Knoten. Jeder Knoten ist über eine Kante mit genau einem Elternknoten verbunden, ausgenommen der Knoten der obersten Ebene, die *Wurzel*. Jeder Knoten repräsentiert ein Datenobjekt, das Attribute wie Höhe (im Baum) oder Anzahl von Kindern besitzen kann. Die Kanten eines Baumes können ebenfalls Attribute besitzen. Eine Kante könnte beispielsweise die Art der Verbindung zwischen seinen beiden Knoten als Attribut besitzen. Eine Aufgabe in einer Baumstruktur ist z.B. das Ermitteln der Höhe des Baumes.

Netzwerk Datentypen der Kategorie *Netzwerk* liegen in einer nichthierarchischen Graphenstruktur vor. Wie in der Baumstruktur stellen Knoten Datenobjekte dar, und sowohl Knoten als auch Kanten können Attribute besitzen. Mögliches Attribut für eine Kante wäre beispielsweise die Länge der Kante. Ein Beispiel für ein Knotenattribut wäre die Distanz zu einem anderen Knoten. Aufgaben in Netzwerken sind u.a. Aufgaben der Graphentheorie, also das Finden von starken Zusammenhangskomponenten, das Finden des kürzesten Weges zwischen zwei Knoten u.ä.

Auf die beschriebenen Datentypen sollen folgende Funktionen angewendet werden können:

Übersicht Eine Visualisierung sollte eine Funkion zu Verfügung stellen, die einen Überblick über den Datensatz liefert. Dazu können alle oder ein Ausschnitt aller Datenobjekte gleichzeitig dargestellt werden. In letzterem Fall sollte ein Mechanismus existieren, um zwischen den dargestellten Bereichen zu wechseln.

Zoom Zoom erlaubt es, eine wählbare Teilmenge des gesamten Datensatzes zu betrachten, beispielsweise nur einen bestimmten Bereich einer Karte. Shneiderman [1996] schlägt verschiedene Techniken vor, darunter das Vergrößern in einer Dimension durch Bedienelemente oder das anklicken eines Datenobjektes, das dann vergrößert dargestellt wird.

Filter Die Filterfunkion soll es dem Benutzer ermöglichen, für ihn nicht relevante Datenobjekte auszublenden oder komplett zu entfernen. Ein Beispiel wäre das Ausblenden aller Gebäude unter drei Stockwerken in einer Karte.

Details-on-demand Bei einer großen Anzahl von Datenobjekten mit vielen Attributen oder assoziierten Informationen müssen sehr viele Informationen gleichzeitig angezeigt werden. Dies reduziert die Übersichtlichkeit stark. Durch Details-on-demand soll der Benutzer erst auf Wunsch nähere Informationen zu einem Datenobjekt oder einer Gruppe von Datenobjekten bekommen. Dadurch kann der Benutzer die Menge an angezeigten Informationen selbst steuern. Ein Beispiel für die Implementierung wären *Tooltips*, wie sie aus vielen Anwendungen bekannt sind.

Verknüpfung Die Verknüpfungsfunktion soll dem Benutzer das Anzeigen von Beziehungen zwischen verschiedenen Datenobjekten erlauben. Dabei könnten beispielsweise Objekte mit gleichen Eigenschaften hervorgehoben werden. Eine Tabelle mit Buchtiteln könnte nach Autoren sortiert werden, um die Beziehung zwischen zwei Buchtiteln hervorzuheben.

History Eine History ist eine Liste der vom Nutzer durchgeführten Aktionen. Diese soll zum Einen erlauben, die Schritte nachzuvollziehen, die zu dieser Ansicht geführt haben („Gefiltert nach Kriterium X, Gefiltert nach Kriterium Y, herangezoomt auf Datenobjekt Z...), zum Anderen, Aktionen rückgängig zu machen. Die „Rückgängig“-Funktion ist aus vielen modernen Anwendungen bekannt und mit der Tastenkombination „Strg-Z“ anwählbar.

Extraktion Das Herausziehen eines gefilterten Datensatzes in eine eigene Umgebung, z.B. das kopieren ausgewählter Tabellenspalten in eine eigene Tabelle. Dies erleichtert die Weitergabe von gefilterten oder bearbeiteten Datensätzen. Eine Form dieser Aktion ist die „Speichern unter...“ Funktion, die von vielen Anwendungen angeboten wird.

Die genannten Funktionen sind eine starke Abstraktion von den tatsächlichen Funkionen, die von einem Benutzer durchgeführt werden. Shneiderman [1996] sieht die Aufgabe der Präzisierung der Funktionen als zukünftige Arbeit. Letztlich ist es die Aufgabe des Entwicklers oder Designers, die konkrete Funktion zu definieren.

Ergebnis der Arbeit ist Shneidermanns „Such-Mantra“: „Overview first, zoom and filter, then details-on-demand.“ ([Shneiderman 1996], S.337). Nach Shneidermann sind die in dem Mantra enthaltenen Funktionen die Kardinalstugenden jeder Visualisierung.

2.1.5 Ereignisbasierte Visualisierung

Tominski [2011] stellt Methoden zur Visualisierung auf der Basis von *Ereignissen* vor. Dabei handelt es sich nicht um eine eigenständige Methode, sondern um eine Erweiterung bestehender Methoden. Die ereignisbasierte Visualisierung ist darauf ausgelegt, für den Nutzer interessante Ereignisse innerhalb eines Datensatzes besonders hervorzuheben. Ein Ereignis ist hier definiert als eine besondere Konstellation von Datenpunkten, beispielsweise wenn ein Wert an drei aufeinanderfolgenden Messpunkten einen bestimmten Schwellwert überschreitet. Welche Ereignisse für den Benutzer interessant sind, kann der Nutzer selbst festlegen, indem er Regeln aufstellt. Zur Erstellung von Regeln stellt Tominski [2011] sowohl eine Syntax als auch ein komplettes Framework vor.

2.2 Verwendete Kriterien

Nachdem im vorangegangenen Abschnitt eine Übersicht über eine Vielzahl an verschiedenen Kriterien gegeben wurde, werden nun die in dieser Arbeit tatsächlich verwendeten Kriterien beschrieben.

Diese Kriterien bilden die Grundlage, anhand derer die in dieser Arbeit beschriebene Visualisierung mit anderen Visualisierungen verglichen werden soll. Die Auswahl wurde nach der angenommenen Wahrscheinlichkeit getroffen, mit der diese für eine gut nutzbare Darstellung der Prozessorauslastung relevant sind. Beispielsweise ist das von Sebrechts et al. [1999] angeführte Kriterium der *Antwortzeit des Systems* auf modernen PCs weit weniger relevant, da mehr als genug Systemressourcen zur Verfügung stehen. Die vollständige Darstellung eines Datensatzes hingegen ist für alle Formen der Visualisierung essentiell.

Weiter wurden, wo möglich, im vorangegangenen Abschnitt 2.1 mehrfach auftretende oder ähnliche Kriterien zusammengefasst. Aus den gefilterten und zusammengefassten Kriterien wurden dann die verwendeten Kriterien gebildet. Die tatsächlich verwendeten

Kriterien sind *Vollständigkeit*, *Granularität*, *Räumliche Aufteilung / Orientierung*, *Animation* und *Aufschlüsselung der Informationen*. Gleichzeitig bilden diese Kriterien den Leitfaden, dem die Entwicklung der konkreten Implementierung (siehe Kapitel 5) folgen soll. Alle vorgestellten Kriterien werden am Ende des Kapitels in Tabelle 1 zusammengefasst.

2.2.1 Vollständigkeit

Erstes Kriterium ist die Vollständigkeit der dargestellten Daten. Vollständigkeit ist hier definiert als die vollständige Darstellung jedes Datums eines Datensatzes, ohne Verkürzung oder Unterlassung. Dies bedeutet im Rahmen dieser Arbeit, dass alle Daten, die von der in Kapitel 5.4 beschriebenen Sonde gesammelt werden, dem Benutzer auch angezeigt werden müssen. Diese sind *Name des ausführenden Threads*, *Name der ausgeführten Methode*, *Verbrauchte Zeit* und *CPU Id*.

Name des ausführenden Threads Um eine Zuordnung von Threads zur Gesamtlaufzeit auf bestimmten logischen Prozessoren vornehmen zu können, muss ein Thread eindeutig identifizierbar sein. Jeder Java-Thread besitzt einen eindeutigen Namen. Dieser kann entweder durch den Entwickler vergeben werden oder automatisch von der JVM erzeugt werden. Die Kieker-Sonde kann diesen Namen auslesen und zur Verfügung stellen (siehe Kapitel 5.4).

Name der ausgeführten Methode (Signatur) Der Name bzw. die Signatur der durch die Kieker-Sonde erfassten Methode. Er ergibt sich aus dem Namen der Java-Klasse und dem tatsächlichen Methodennamen (z.B. „Integer.toString()“). Die Signatur wird benötigt, um die *Hungriest Method* ermitteln zu können, also die Methode, die innerhalb eines Threads am häufigsten aufgerufen wurde.

Verbrauchte Zeit (Laufzeit) Vor und nach Ausführung einer durch die Kieker-Sonde erfassten Methode wird die Systemzeit ausgelesen. Aus der Differenz zwischen den beiden Zeitstempeln ergibt sich die *Laufzeit*, die diese Methode zur Ausführung benötigt hat. Diese Zeiten werden für jeden Thread separat gespeichert. Zusammen mit der ID des logischen Prozessors und dem Namen des Threads, auf dem die Methode ausgeführt wurde, lässt sich näherungsweise die Gesamtzeit ermitteln, die ein Thread auf einem bestimmten logischen Prozessor verbracht hat.

CPU Id Ein zentraler Punkt der Messung ist die Ermittlung des logischen Prozessors, auf dem der Thread zu Beginn der Messung (vor Aufruf der Methode) und danach (nach der Rückkehr der Methode) ausgeführt wurde. Ähnlich wie ein Thread ist auch ein logischer Prozessor innerhalb eines Systems eindeutig identifizierbar (siehe Kapitel 5.1.2). Es ist möglich, dass ein Thread zwischen zwei Messpunkten den logischen Prozessor wechselt. Daher müssen die beiden Messungen nicht dieselbe CPU Id liefern (siehe Kapitel 5.6.1).

Dabei sind die Kerndaten des Datensatzes definiert als der Name des Threads sowie seine bisherige Laufzeit auf dem logischen Prozessor zu einem bestimmten Zeitpunkt.

Alle weiteren Daten sind „Kontextinformationen“ die zur Analyse hilfreich, aber nicht entscheidend sind. Dabei können Daten auch optional anwählbar sein (siehe Kap. 2.2.2 weiter unten).

Dieses Kriterium berücksichtigt auch die von Freitas et al. [2002] angesprochene Vollständigkeit des „semantischen Inhalts“, also das Darstellen von Beziehungen innerhalb eines Datensatzes. So ist eine Visualisierung, die alle oben genannten Daten anzeigt, trotzdem unvollständig, wenn nicht ersichtlich ist, auf welchem logischen Prozessor ein Thread ausgeführt wird. „Semantische Vollständigkeit“ beinhaltet auch die explizite Darstellung von Information, also aus Daten gewonnenem Wissen. Beispielsweise könnte ein Thread, der einen Kern dauerhaft vollständig auslastet, als spezieller „Unruhestifter“ markiert werden.

Das Kriterium der Vollständigkeit kann sowohl „hart“ als auch „weich“ ausgelegt werden. Bei einer harten Auslegung dieses Kriteriums müssen alle genannten Daten durch die Visualisierung selbst angezeigt werden können, ohne zusätzliche Hilfen wie Textboxen. Beschriftungen (Label) innerhalb der Visualisierung fallen nicht in diese Kategorie. Eine weiche Auslegung bedeutet, dass nur die Kerndaten durch die Visualisierung dargestellt werden müssen, Kontextdaten jedoch neben der eigentlichen Visualisierung dargestellt werden dürfen, beispielsweise in Form einer Textbox oder Tabelle.

Dieses Kriterium basiert auf dem Kriterium *Vollständigkeit* von Freitas et al. [2002] und schließt die von Shneiderman [1996] vorgestellte Funktion *Verknüpfung* mit ein.

2.2.2 Granularität

Ein der Vollständigkeit verwandtes Kriterium ist die Granularität der Darstellung. Eine tatsächlich vollständige Darstellung, wie im obigen Kapitel 2.2.1 beschrieben, wird gerade bei größeren Datensätzen unübersichtlich. Man stelle sich ein komplettes, mehrjähriges Systemlog vor, das komplett den Bildschirm füllt. Allein aufgrund der zu wählenden Schriftgröße wäre eine für den Benutzer nützliche Darstellung nicht zu erreichen, die Menge an Daten würde jedes für den Benutzer interessante Detail erschlagen. Aus diesem Grund sollte die Granularität der Darstellung wählbar sein. In unserem Systemlog-Beispiel sollte also in der Übersicht über das Log nicht der komplette Text angezeigt werden, sondern z.B. zunächst nur eine Liste mit Jahresangaben. Das Anwählen der Jahresangaben führt zu einer Unterteilung in Monate, hier vielleicht schon mit zusätzlichen Informationen wie die durchschnittliche Prozessorlast in diesem Monat. Die Anwahl eines Monats könnte nun die tatsächlichen Logeinträge zeigen. Zusätzlich sollte der Benutzer auf jedem Level die Möglichkeit haben, die Einträge weiter nach eigenen Angaben zu Filtern, beispielsweise nach einer bestimmten Kategorie von Einträgen (z.B. „Prozessorlast“) oder nach frei wählbaren Schlüsselworten innerhalb der Einträge („Auslastung $>$ 100%“).

Im Rahmen Visualisierung der Prozessorauslastung bedeutet Granularität hauptsächlich die Eingrenzung der dargestellten Zeitspanne. Möglich sind beispielsweise eine Zu-

sammenfassung über einen bestimmten Zeitraum oder die Betrachtung eines bestimmten Zeitpunktes. Zudem kann auch die Granularität der Ebene als Aspekt herangezogen werden, also ob ein Überblick des gewählten Zeitraums auf Methoden- Thread- oder Prozessorebene möglich ist.

Dieses Kriterium fasst die von Shneiderman [1996] aufgestellten Aufgaben „Zoom“, „Overview“, „Details-on-demand“ und „Filter“ zusammen. Auch das Kriterium „Verkleinerung des Datensatzes“ von Freitas et al. [2002] fällt hier hinein.

2.2.3 Räumliche Aufteilung / Orientierung

Um eine große Menge an Daten verschiedener Art schnell erfassen zu können, ist es notwendig, den zur Darstellung der Daten verfügbaren Platz zu gliedern. Bei der Aufteilung der Anzeigefläche in Bereiche und Unterbereiche sollten zusammenhängende Daten (z.B. Zeitstempel und Logeintrag) nahe beieinander dargestellt werden. Die weitere Aufteilung sollte sich an den Eigenschaften der dargestellten Daten orientieren [Freitas et al. 2002]. So können textuell dargestellte Daten, beispielsweise Details zu einem graphisch dargestellten Messpunkt, als zusätzliche Box neben der graphischen Darstellung angeordnet sein oder als Beschriftung in die graphische Darstellung integriert sein. In Erweiterung des Kriteriums „Granularität“ (siehe Kap. 2.2.2) könnte diese Anzeige auch ein- und ausblendbar sein. Optionale Anzeigen sollten bei der Gestaltung der räumlichen Aufteilung berücksichtigt werden. Sie sollten, wenn eingeblendet, keine wichtigen Informationen verdecken.

Der Anwender sollte die Möglichkeit haben, die für ihn relevanten Daten mittig darzustellen oder einen nach seinen Kriterien gefilterten Datensatz so anzuordnen, wie es für ihn die beste Übersicht bietet. Dadurch können dem Anwender zuvor verborgene Zusammenhänge zwischen den Daten auffallen und zusätzliche Informationen gewonnen werden.

Nach Shneiderman [1996] ist bei der räumlichen Aufteilung zwischen 1-, 2-, 3-, und multi-dimensionalen Darstellungen zu unterscheiden. Jede der Darstellungsformen eignet sich unterschiedlich gut zur Darstellung bestimmter Daten. Vor- und Nachteile der Darstellung in verschiedenen Dimension wiegen sich jedoch nach Pastizzo et al. [2002] weitgehend auf.

Dieses Kriterium bezieht sich nicht nur auf die Visualisierung der Daten, sondern auch auf Bedienelemente, die eine Veränderung der dargestellten Daten oder der Darstellungsart ermöglichen. Beispiele für Bedienelemente wären Buttons zum Auslösen von Aktionen oder Schiebregler zum Auswählen eines Zeitpunktes oder -raumes.

2.2.4 Animation

Bei zeitbasierten Visualisierungen, die schrittweise verschiedene Zustände hintereinander darstellen, wird ein Mechanismus für die Übergänge zwischen den Zuständen benötigt.

Im Sinne dieses Kriteriums besteht zum einen die Möglichkeit, übergangslos von einem Schritt zum nächsten zu wechseln, zum anderen, flüssige Übergänge zwischen den Zuständen zu schaffen. Flüssige Animationen erleichtern das Wiederfinden und Erkennen von beobachteten Objekten. Wird der Zustandsübergang eines Objektes zwischen zwei Schritten sprunghaft dargestellt, so muss der Anwender das im Moment betrachtete Objekt erst suchen, da es den Anschein hat, das beobachtete Objekt wäre verschwunden und ein neues Objekt an seine Stelle getreten. Wird dieser Zustandsübergang animiert, bleibt der Eindruck eines eindeutigen Objekts erhalten, da sich das beobachtete Objekt lediglich verändert. Diese Technik wird auch bei animierten Kontextmenüs verwendet.

Im Rahmen der Visualisierung von Prozessorauslastung bedeutet dies, dass die Veränderung des Auslastungswertes von Schritt zu Schritt flüssig erfolgen sollte. Insbesondere sollte der Wechsel eines Threads von einem logischen Prozessor zu einem anderen nicht übergangslos dargestellt werden. Kamerabewegungen, sowohl im zwei- als auch im dreidimensionalen Raum, sollten nicht ruckartig erfolgen, da dies die Orientierung für den Anwender erschwert.

Dieses Kriterium ist abgeleitet von den Kriterien „Zustandsübergänge“, „Orientierung und Hilfe“ (beide Freitas et al. [2002]) und „Räumliche Orientierung“ (Sebrechts et al. [1999]).

2.2.5 Aufschlüsselung der Informationen / Farbkodierung

Visualisierungen haben die Aufgabe, Informationen möglichst kurz und übersichtlich zu vermitteln. Innerhalb einer Visualisierung gibt es verschiedene Wege, Informationen zu vermitteln. Ein Weg ist, verschiedene Symbole für häufig auftretende Informationen zu nutzen oder bestimmten Farben oder Positionen im Raum eine Bedeutung zu geben (beispielsweise Rot als Warnfarbe oder die mittige Darstellung eines besonders wichtigen Datums). Dieses Kriterium beschreibt die Aufschlüsselung der Informationen, also die Art und Weise, wie ein bestimmtes Datum dargestellt wird. Ein Datum kann beispielsweise textuell (z.B. "23") oder, wie in einer *Heatmap* (siehe Kap. 3.4) durch einen passenden Farbwert dargestellt werden. Die räumliche Anordnung eines Datums kann eine Bedeutung haben, beispielsweise in einem *Graphen*, wo das Datum durch die Position innerhalb des Graphen ausgedrückt wird. Die Bedeutung bestimmter, speziell der neu eingeführten, Symbole muss dem Anwender verständlich sein. Um dem Anwender die Bedeutung der dargestellten Symbole zu vermitteln kann beispielsweise eine (einblendbare) Legende benutzt werden.

Im Rahmen dieser Arbeit bezieht sich das Kriterium „Aufschlüsselung der Informationen“ auf die Darstellung der Kerninformationen, also Threadname und Laufzeit auf dem momentanen logischen Prozessor sowie die Möglichkeiten, bestimmte Informationen hervorzuheben. Dieses Kriterium beruht auf den Kriterien „Aufschlüsselung der Informationen“ ([Freitas et al. 2002] und „Farbkodierung von Eigenschaften“ ([Sebrechts et al. 1999]).

Name	**Zusammenfassung**
Vollständigkeit	Werden alle verfügbaren Daten angezeigt? Sind Verknüpfungen sichtbar?
Granularität	Ist der Detaillevel / Menge der angezeigten Informationen wählbar?
Räumliche Aufteilung / Orientierung	Findet der Benutzer schnell, was er sucht? Wird ihm Hilfe bei der Orientierung geboten?
Animation	Sind Objekte nach einem Zustandsübergang noch erkennbar?
Aufschlüsselung der Informationen	Sind alle Informationen so dargestellt, das der Benutzer sie erkennen kann?

Tabelle 1: Verwendete Kriterien (Zusammenfassung)

3 Bestehende Visualisierungen

Im vorhergehenden Kapitel haben wir Kriterien zusammengetragen, anhand derer eine Einordnung und rudimentäre Bewertung der Eignung von Visualisierungstechniken vorgenommen werden kann. In diesem Kapitel werden diese Kriterien auf einige bereits bestehende Visualisierungen angewendet. Da die Anzahl der Visualisierungen, die direkt auf die Darstellung von Prozessorauslastung ausgerichtet sind, gering ist, sind hier hauptsächlich generelle Visualisierungstechniken beschrieben. Eine Ausnahme dazu ist ParaGraph, das am Ende dieses Kapitels beschrieben wird (siehe Kapitel 3.7).

Die Auswahl der Visualisierungstechniken soll den in dieser Arbeit vorgestellten Ansatz in einen Kontext setzen und einen Vergleich ermöglichen. Wie Sebrechts et al. [1999] beschreiben, ist ein Vergleich von 2D- und 3D-Techniken erlaubt, solange die angebotenen Funktionen ähnlich oder identisch sind. Daher sind hier auch 2D-Visualisierungen aufgeführt.

Zu jeder der gewählten Visualisierungen wird eine zunächst kurze Beschreibung gegeben. Als nächstes folgt eine Diskussion der Visualisierung anhand der Kriterien aus Kapitel 2.2. Zu jeder Visualisierung wird ein bereits existierendes Beispiel als Grundlage für einen theoretischen Vergleichsansatz genommen. Um untereinander vergleichbare Ansätze zu erhalten, muss diese Grundlage gegebenenfalls erweitert werden. Die Erweiterungen werden in der jeweiligen Diskussion beschrieben.

3.1 Textuelle Darstellung

Eine der einfachsten Arten der Darstellung von Daten ist die textbasierte Form. Dies ist die „natürliche“ Form der Darstellung für Textdokumente oder auch Quellcode.

Oft werden Datensätze auch zu Listen oder Tabellen zusammengefasst. Eine einzelne Eigenschaft eines Datenobjektes wird dabei in einer Spalte der Tabelle dargestellt. Auch die Darstellung mehrerer Tabellen für verschiedene Eigenschaften ist möglich, falls ausreichend Platz zur Verfügung steht. In diesem Fall werden die einzelnen Zeilen einer Tabelle meist mit einem eindeutigen Schlüssel indiziert, um die Verknüpfung von Daten über die Tabellen hinweg zu ermöglichen.

Ein Beispiel aus dem Bereich der Darstellung von Prozessorauslastung wäre das Linux-Tool *top* (siehe Abb. 1). top listet alle momentan auf dem System laufenden Prozesse zusammen mit zusätzlichen Informationen zu den Prozessen (z.B. ProzessId, momentaner Speicherverbrauch und erzeugte Prozessorlast) auf. Die zusätzlich angezeigten Informationen sowie die Farbwahl können vom Benutzer konfiguriert werden. Im oberen Teil der Anzeige werden allgemeine Informationen über das System angezeigt (z.B. uptime und gesamte Prozessorauslastung über die Zeit).

```
top - 15:27:00 up 166 days,  7:48,  1 user,  load average: 0.28, 0.18, 0.11
Tasks: 183 total,   1 running, 182 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.1%us,  0.1%sy,  0.0%ni, 99.8%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8172576k total,  8092160k used,    80416k free,   513156k buffers
Swap:  4194296k total,      116k used,  4194180k free,  6221044k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
16325 root      15   0 10884 1120  784 R  0.3  0.0   0:00.09 top
30158 named     21   0  251m 7128 2088 S  0.3  0.1   0:06.48 named
    1 root      15   0 10372  696  580 S  0.0  0.0   0:19.67 init
    2 root      RT  -5     0    0    0 S  0.0  0.0   0:47.54 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:02.56 ksoftirqd/0
    4 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/0
    5 root      RT  -5     0    0    0 S  0.0  0.0   0:57.06 migration/1
    6 root      34  19     0    0    0 S  0.0  0.0   0:03.89 ksoftirqd/1
    7 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/1
    8 root      RT  -5     0    0    0 S  0.0  0.0   5:10.46 migration/2
    9 root      34  19     0    0    0 S  0.0  0.0   0:56.67 ksoftirqd/2
   10 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/2
```

Abbildung 1: top

Ansatz im Rahmen der Prozessorauslastung

Als Basis für einen Beispielansatz wird das oben beschriebene top verwendet. top bietet bereits viele Funktionen, so dass nur wenige Erweiterungen gemacht werden müssen.

Vollständigkeit Eine textuelle Darstellung kann grundsätzlich jedes Datum, das durch Text codiert werden kann, darstellen. Da praktisch alle Daten textuell codiert werden können, können insbesondere auch die für eine vollständige Darstellung im Sinne dieser Arbeit notwendigen Daten dargestellt werden. Damit ist eine textuelle Darstellung automatisch vollständig. top bietet für die Anzeige der (gesamten) verbrauchten Prozessorzeit, des zuletzt genutzten logischen Prozessors und des Thread- bzw. Prozessnamens bereits Schalter an. Die Anzeige der ausgeführten Methode wird nicht von top unterstützt, ist aber theoretisch leicht zu einer textuellen Darstellung hinzuzufügen.

Die semantische Vollständigkeit kann über die räumliche Aufteilung und geschickte Wahl von Tabellen erreicht werden.

Granularität top bietet dem Benutzer bereits die Möglichkeit, die angezeigten Tabellenspalten zu konfigurieren. Weiter ist es möglich, Filter für die angezeigten Prozesse oder Threads zuzuschalten. Diese erlauben jedoch beispielsweise keine Filterung nach Schwellwerten was jedoch (für unseren theoretischen Vergleichsansatz) leicht zu ergänzen ist. Auch eine Sortierung der Tabelle nach verschiedenen Spalten ist bereits möglich. Grundsätzlich können textuelle Darstellungen Daten nahezu beliebig fein oder grob darstellen. Die Grenzen liegen dabei nicht in der Darstellung selbst, sondern in den angebotenen Filtern.

Räumliche Aufteilung und Orientierung Die räumliche Aufteilung von top ist schlicht gehalten, das Hauptaugenmerk liegt auf der Tabelle der angezeigten Prozesse / Threads.

Aufgrund der einfachen Natur der textuellen Darstellung ist der Grad der Anpassbarkeit der Anordnung begrenzt. Eine freie Anordnung von Informationen ist aufgrund des hohen Aufwands beim generieren eines Layouts selten möglich. top bietet als reine Konsolenanwendung keine graphischen Bedienelemente, es kann jedoch in einen Konfigurationsmodus gewechselt werden, in dem allerdings keine Prozess- oder Threaddaten angezeigt werden.

Im Rahmen des Vergleichsansatzes kann die Tabelle in ein Bedienfeld eingebettet werden. Dazu werden unter oder neben der Tabelle Bedienelemente angeordnet, mit denen die Tabelle gefiltert oder ihre Spalten konfiguriert werden können.

Animation top bietet keine Animationen. Die Änderungen der angezeigten Werte und der Sortierungen erfolgen Sprunghaft. Zwar ist das Animieren der Wertübergänge denkbar, bietet jedoch keinen Vorteil, da die Überblendung zwischen zwei Werten am gleichen Platz geschehen würde und damit der Wert selbst unleserlich würde. Findet die Werteveränderung nicht am gleichen Platz statt, beispielsweise bei Fortschrittsbalken, sind Animationen in begrenztem Maße möglich und vorteilhaft.

Aufschlüsselung der Symbole und Farben Die Darstellung der Kerninformationen erfolgt, naturgemäß, textuell. Dabei können, wenn gewünscht, bestimmte Informationen durch Sonderzeichen oder Abkürzungen dargestellt werden, um Platz zu sparen. Sollte genug Platz zur Verfügung stehen, ist es jedoch ratsam, eine „sprechende“ Darstellung zu wählen, beispielsweise ein vorangestelltes „(aktueller Thread)“ statt eines einzelnen Zeichens wie „$“.

Textuelle Darstellungen bieten, neben der räumlichen Aufteilung, eine Reihe anderer Möglichkeiten zur Hervorhebung von Daten, darunter das Verändern der Text- oder Hintergrundfarbe, das Ändern der Schriftart oder Schrifttyps oder das Verändern des Abstands zwischen zwei Worten oder Zeilen.

Fazit Eine textuelle Darstellung der Prozessorauslastung bietet den Vorteil einer schnellen Erlernbarkeit, da alle Informationen ohne ungewöhnliche Kodierung auskommen können. Ein weiterer großer Vorteil ist die einfache Realisierung einer textuellen Darstellung. Es ist für eine Implementierung nicht nötig, eine Grafikbibliothek zu bemühen.

Nachteil der textuellen Darstellung ist die fehlende Möglichkeit, sich einen schnellen Überblick über alle Datensätze zu verschaffen, wie es z.B. mit einem Funktionsgraphen (siehe Kap. 3.2) möglich ist.

3.2 Funktionsgraph

Funktionsgraphen sind eines der Standardwerkzeuge in der Visualisierung von Daten. Die Einfachheit und die weite Verbreitung dieser Visualiserungstechnik machen sie für eine Verwendung zur Analyse der Prozessorauslastung interessant.

Ein Funktionsgraph besteht gewöhnlich aus einem zweidimensionalen, in manchen Fällen auch dreidimensionalen, Koordinatensystem, dessen Achsen mit jeweils einer mess-

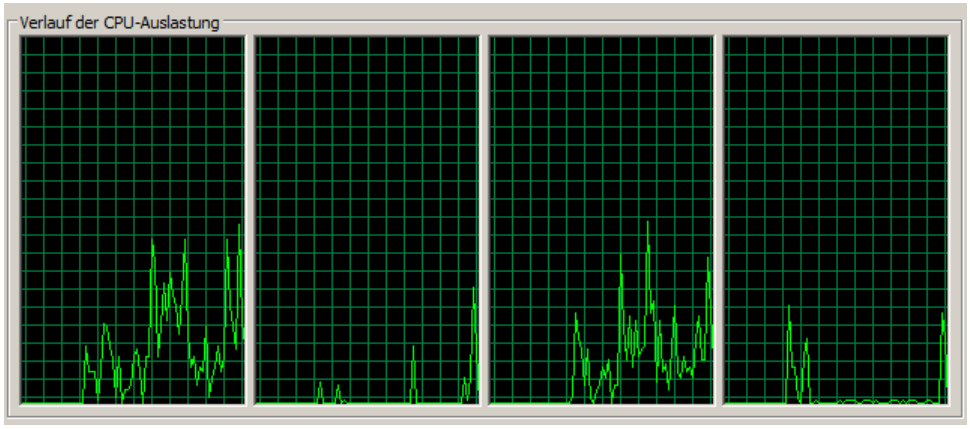


Abbildung 2: Windows 7 Taskmanager.

baren Eigenschaft beschriftet sind. Ein Datum wird in Form eines Punktes dargestellt. Die Koordinaten des Punktes ergeben sich aus den Werten des Datums für die auf den Achsen angegebenen Eigenschaften. Häufig wird die X-Achse als Zeitachse genutzt, so dass eine Entwicklung der Eigenschaft auf der Y-Achse über die Zeit betrachtet werden kann (beispielsweise die prozentuale Auslastung eines logischen Prozessors).

Als Darstellung der Prozessorauslastung über die Zeit werden Funktionsgraphen beispielsweise unter Windows (seit Windows 95) verwendet. Abbildung 2 zeigt die Graphendarstellung der Prozessorauslastung eines Systems mit vier logischen Prozessoren. Auf der X-Achse ist die Zeit aufgetragen, auf der Y-Achse die prozentuale Auslastung des logischen Prozessors.

Ansatz im Rahmen der Prozessorauslastung

Als Basis des funktionsgraphenbasierten Ansatzes wird der oben genannte Windows-Taskmanagers genommen. Um einen zu den anderen Ansätzen und dem Prototypen vergleichbaren Ansatz zu erhalten, muss diese Darstellung erweitert werden.

Vollständigkeit Mit dem vorgestellten Ansatz ist es möglich, einen Datensatz vollständig im (weichen) Sinne von Kapitel 2.2.1 darzustellen. Dazu müssen zusätzliche Informationen wie die ID des logischen Prozessors und seines physikalischen Prozessors für jeden der Funktionsgraphen angegeben werden. Dies kann beispielsweise durch eine Bildunterschrift oder eine zusätzliche Informationsbox geschehen.

Um mehrere Threads darzustellen, muss für jeden logischen Prozessor ein Funktionsgraph mit mehreren aufgetragenen Kurven angezeigt werden. Die zu einem bestimmten Zeitpunkt auf einem Thread ausgeführte Methode kann angezeigt werden, indem der Bereich der Kurve, die dem Ausführungszeitraum entspricht, mit dem Namen der Methode

markiert wird. Zur Unterscheidung zweier nebeneinander liegender Zeiträume können die Bereiche durch unterschiedliche Helligkeiten der Kurve voneinander abgesetzt werden.

Besonders die in in 2.2.1 erwähnte „semantische Vollständigkeit“ kann durch einen Funktionsgraphen erreicht werden, da die Beziehung zwischen momentaner Prozessorlast und Zeit hervorgehoben wird. Beides wird auf den Achsen des Funktionsgraphen aufgetragen und damit direkt miteinander in Beziehung gesetzt.

Granularität Durch Ein- und Ausblenden nicht benötigter Daten oder Verschieben des betrachteten Bereichs auf einer der Achsen kann die Granularität der Darstellung verändert werden. Denkbar ist auch eine Veränderung des Maßstabs der Achsen, um feinere Veränderungen betrachten zu können oder einen weniger detaillierten Überblick zu gewinnen.

Aufschlüsselung der Symbole und Farben Die Aufschlüsselung der Daten eines Funktionsgraphen geschieht für die zentralen Daten (Auslastung und Messzeitpunkt) über die Achsenbeschriftungen.

Verschiedene Threads können durch unterschiedlich eingefärbte Kurven dargestellt werden. Denkbar ist auch eine Hervorhebung der Maxima der Kurven durch spezielle Symbole. Dazu kann optional eine Legende eingeblendet werden.

Räumliche Aufteilung und Orientierung Es bietet sich an, als zentrales Element der Darstellung den Funktionsgraphen selbst darzustellen. Die Informationsbox sollte in einer entsprechenden Breite seitlich dazu angeordnet werden.

Animation Die Funktionsgraphen könnten schrittweise von Messpunkt zu Messpunkt animiert werden. Dabei könnte der betrachtete Bereich um den neuen Messzeitpunkt erweitert werden und anschließend die Kurven nachgezeichnet werden.

Fazit Der graphische Teil der Visualisierung, der Funktionsgraph, kann nur ein Teil einer vollständigen Darstellung der Prozessorauslastung sein. In Kombination mit zusätzlichen, textuell dargestellten Informationen jedoch sind Funktionsgraphen ein hilfreiches Werkzeug zur Analyse. Ein großer Vorteil des Funktionsgraphen ist dabei die starke Hervorhebung des Zusammenhangs zwischen Prozessorauslastung und Messzeitpunkt, die bei der Analyse einer Anwendung meist von großem Interesse sind. Ein weiterer Vorteil ist die weite Verbreitung dieser Darstellungsform, so dass sich die Eingewöhnungsphase für den Anwender verkürzt. Zudem ist mit Funktionsgraphen sowohl eine schrittweise Betrachtung der Auslastung möglich, als auch das Betrachten des gesamten Messzeitraumes. Als Nachteil kann gewertet werden, das die Darstellung von mehr als zwei Achsen unübersichtlich bis unlesbar werden kann. Eine Verwendung von mehr als drei Achsen sollte nicht in Betracht gezogen werden, da dies nur über Projektionen und Umrechnungen geschehen kann, die für Menschen nicht übersichtlich oder einfach zu erfassen sind. Dieser Nachteil ist jedoch im Rahmen der Analyse der Prozessorauslastung nur von geringer Bedeutung, da hier zwei Achsen zur Darstellung der Kerninformationen ausreichen.

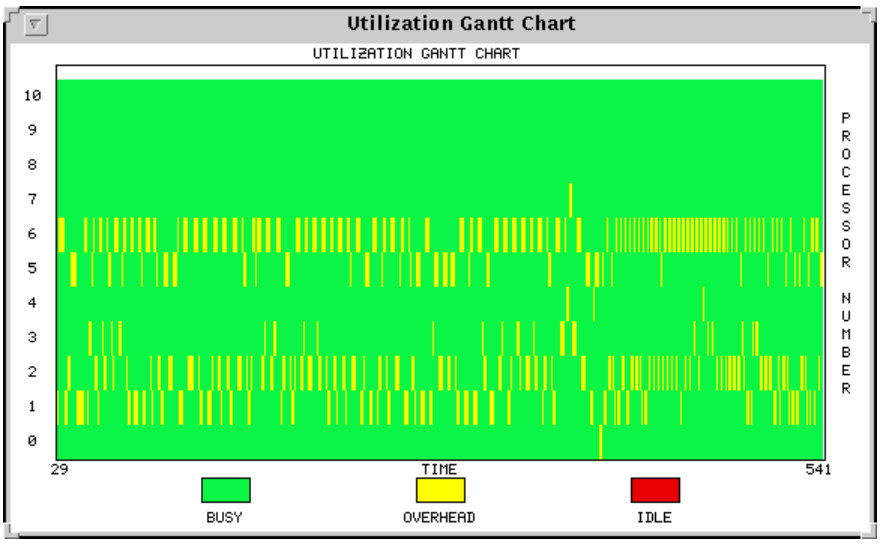


Abbildung 3: ParaGraph Gantt-Diagramm. Quelle: Xu and Robbins [1996]

3.3 Gantt-Diagramm

Ein Gantt-Diagramm, benannt nach seinem Entwickler Henry Gantt, ist eine Variante eines Balkendiagramms. Es wird für gewöhnlich in der Zeit- und Projektplanung genutzt. In einem Gantt-Diagramm wird die geplante Zeiteinteilung einer Aufgabe als horizontaler Balken neben der Bezeichnung der Aufgabe aufgetragen. Als Wertebereich des horizontalen Balkens wird dabei ein bestimmter Zeitraum gewählt, vom Beginn des Projekts bis zum voraussichtlichen Ende. Anstatt einer Aufgabe kann auch eine Ressource, deren Verfügbarkeit geplant werden soll, eingesetzt werden. Heath and Finger [2003] beschreiben den Einsatz von Gantt-Diagrammen im Rahmen der Visualisierung von Prozessorauslastung.

Ansatz im Rahmen der Prozessorauslastung

Als Basis des Vergleichsansatzes wird der Ansatz von Heath and Finger [2003] genommen (siehe Abb. 3). Dieser Ansatz ist darauf ausgelegt, lediglich den Prozessorstatus zu visualisieren (eine Unterscheidung zwischen „busy“, „idle’ und „overhead“). Um einen zu den anderen Visualisierungen vergleichbaren Ansatz zu schaffen, kann er leicht verändert werden.

Vollständigkeit Um Vollständigkeit im Sinne von Kap. 2.2.1 zu erreichen, muss zunächst die Anzeige des ausführenden Threads in das Diagramm eingebaut werden. Dabei können wir uns, wie der Basisansatz, der Möglichkeit bedienen, die Balken des Dia-

gramms unterschiedlich einzufärben. Jedem Thread kann eine eindeutige Farbe zugeordnet werden. Anschließend können die Balken entsprechend eingefärbt werden, also für den entsprechenden logischen Prozessor zur entsprechenden Zeit die Farbe des ausgeführten Threads. Dadurch wird gleichzeitig die Zuordnung von einem Thread zur Laufzeit auf einem logischen Prozessor angezeigt.

Als nächstes kann eine Anzeige der momentan ausgeführten Methode hinzugefügt werden. Weiter kann ein Schiebregler über der Zeitachse hinzugefügt werden, der die Auswahl eines einzelnen Zeitpunktes erlaubt. Der Schiebregler schneidet dabei die Balken, ähnlich der Frequenzwahl alter Radios. Am Schnittpunkt zwischen Schiebregler und Balken wird der Name der zu diesem Zeitpunkt ausgeführten Methode angezeigt. Alternativ kann diese Anzeige auch für andere Kontextdaten genutzt werden.

Granularität Ähnlich wie bei einem Time Wheel (siehe Kap.3.5) kann die Granularität der Darstellung hauptsächlich über die Wahl des dargestellten Zeitraumes bestimmt werden. Weitere Methoden der Kontrolle der Granularität wären das Ein- und Ausblenden des Schiebreglers.

Räumliche Aufteilung und Orientierung Der Basisansatz, wie in Abb. 3 dargestellt, sollte um klare Linien und Grenzen erweitert werden. Zunächst sollten die einzelnen Balken horizontal getrennt werden, um ein Verrutschen zwischen den Balken zu vermeiden. Weiter ist es für unser Szenario notwendig, die Wechsel zwischen den Threads farblich hervorzuheben. Im Basisansatz werden nur drei Zustände durch rot, gelb und grün dargestellt. Diese bieten von sich aus einen guten Kontrast zueinander, was eine Abgrenzung nicht notwendig macht. In unserem erweiterten Ansatz müssen wir eine unbestimmte Anzahl von Threads unterscheiden können. Da die Farbgebungen zweier Threads unter Umständen sehr ähnlich sind, könnte die Unterscheidbarkeit von nicht beeinflussbaren Umständen wie Kontrast und Qualität des Anzeigegerätes oder der Farbwahrnehmung des Anwenders abhängen. Daher sollte der Wechsel durch eine dünne Linie in einer neutralen Farbe, beispielsweise Weiß, hervorgehoben werden, um beide Threads klar voneinander zu trennen.

Alternativ zur horizontalen Anordnung der Zeitachse können die Balken auch vertikal aufgetragen werden. In diesem Fall erhält man ein gestapeltes Säulendiagramm.

Animation Ein Gantt-Diagramm kann entlang der Zeitachse schrittweise animiert werden. Die Bewegungen des Schiebreglers sollten ebenfalls weich animiert und möglichst fein aufgelöst sein, um dem Anwender das Anwählen bestimmter Zeitpunkte zu erleichtern.

Aufschlüsselung der Symbole und Farben Zur Visualisierung der Prozessorauslastung müssen in einem Gantt-Diagramm, abgesehen von den Balken selbst, wenige zusätzlichen Symbole definiert werden. Die Bedeutung der Balken ist leicht zu erfassen. Es sollte ein deutlicher visueller Hinweis auf die Funktion des Schiebreglers geschaffen werden, beispielsweise durch seine Gestaltung. Die Farbgebung der einzelnen Threads bedarf ebenfalls einer Erklärung dem Anwender gegenüber. Dies kann entweder durch eine Legende geschehen, oder, um die Vollständigkeit zu erhalten, durch eine Beschriftung mit dem Threadnamen innerhalb der Balken. Dies kann allerdings bei schnellen Threadwechseln

zu Unleserlichkeit führen, da dann die einzelnen Abschnitte zu kurz für die Beschriftung sein könnten.

Fazit Gantt-Diagramme haben in anderen Bereichen, beispielsweise in der Projektplanung, weite Verbreitung gefunden. Anwender sind daher bereits mit der Darstellungsform vertraut. Dadurch sind die Diagramme auch für unerfahrene Anwender leicht lesbar und verständlich. Nachteil von Gantt-Diagrammen ist, dass sie bei größeren Mengen an logischen Prozessoren an Übersichtlichkeit verlieren, da eventuell nicht mehr alle logischen Prozessoren gleichzeitig oder nur noch in schwer zu erkennbarer Größe dargestellt werden. Anstatt eines Gantt-Diagramms können auch andere Varianten von Balkendiagrammen verwendet werden.

3.4 Heatmap

Heatmaps sind eine Kategorie von Visualisierungen, die Werte auf einer zweidimensionalen Ebene farbkodiert darstellen.

Namensgebend sind Wärmebilder, wie sie von Wärmebildkameras gemacht werden. Hier werden an verschiedenen Punkten gemessene Temperaturen in verschiedenen Farben codiert (z.B. Blau für „kalt“ und Rot für „warm“) und über das Bild des gemessenen Objekts gelegt.

Ein Beispiel für eine Heatmap im Zusammenhang mit der Visualisierung der Prozessorauslastung ist der Taskmanager von Windows 8 (siehe Abb. 4). Die Abbildung des Taskmanagers zeigt ein Gitter aus Icons, in dem jedes Icon einen logischen Prozessor repräsentiert. Zusätzlich zu der prozentualen Auslastung des Kerns wechselt das Icon die Farbe in Abhängigkeit der Prozessorauslastung.

Ansatz im Rahmen der Prozessorauslastung

Wie bereits für die Funktionsgraphen in Kap. 3.2 wird als Basis des Ansatzes der bestehende Ansatz des Windows 8 Taskmanagers erweitert.

Vollständigkeit Um einen vollständigen Datensatz mithilfe einer Heatmap darzustellen, müssen zunächst die Daten für einzelne Threads dargestellt werden. Dabei müssen die Threads mit den zugehörigen logischen Prozessoren verbunden werden. Eine mögliche Lösung dafür ist, die vorhanden Icons der logischen Prozessoren weiter zu unterteilen. In den dadurch gewonnenen Sub-Icons können dann die Auslastungen der einzelnen Threads auf die gleiche Weise dargestellt werden wie zuvor die Gesamtauslastung, also textuell und durch Farbkodierung. Jedes der Sub-Icons muss zusätzlich den Namen des mit ihm verbundenen Threads bereitstellen. Dazu reicht eine textuelle Darstellung innerhalb eines Icons zusätzlich zur Auslastung.

Räumliche Aufteilung und Orientierung Damit durch die zusätzlichen Informationen die Lesbarkeit der Threadnamen und des Auslastungswerts erhalten bleibt, muss die gesamte Darstellung vergrößert werden. Dadurch benötigt die Heatmap schnell mehr Platz

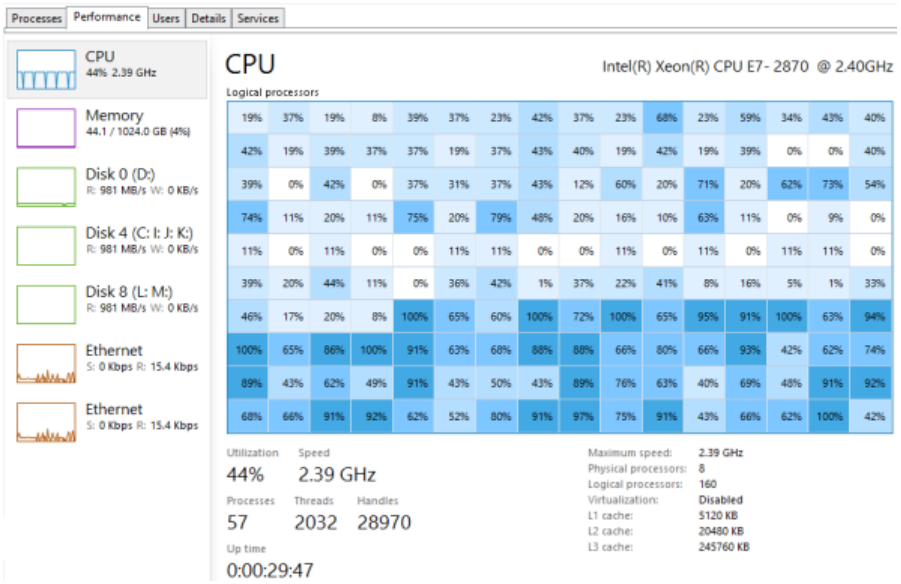


Abbildung 4: Windows 8 Taskmanager. Quelle: Hruska [2011]

als für die Darstellung zur Verfügung steht. Mehr dazu im nächsten Abschnitt.

Wie bereits in Abbildung 4 zu sehen, können neben der graphischen Visualisierung zusätzliche Informationen eingeblendet werden.

Granularität Wie im letzten Absatz beschrieben, kann die Größe der Heatmap schnell den zur Verfügung stehenden Platz überschreiten. Um weiterhin Zugriff auf alle Informationen erhalten zu können, wird der Benutzer die Heatmap entweder horizontal und vertikal verschieben oder die Ansicht verkleinern müssen. Im ersteren Fall sollte eine Art „Minimap“ eingeblendet werden, die Aufschluss über die Position des momentan betrachteten Bereichs gibt. Beispielsweise könnte innerhalb der Minimap ein Rahmen um den dargestellten Bereich gezeichnet werden.

Im Falle einer Verkleinerung / Vergrößerung sollte nur eine stufenweise Veränderung der Größe erlaubt werden. Dadurch kann zwischen verschiedenen Detailgraden der Darstellung gewechselt werden. Die Heatmap könnte beispielsweise in der kleinsten Darstellung nur dieselben Informationen anzeigen wie die ursprüngliche Darstellung, also die allgemeine Auslastung der logischen Prozessoren.

Animation Für Heatmaps bietet sich ein sanfter Übergang der Iconfarben zwischen den einzelnen Messzeitpunkten an. Wird eine Darstellung mit verschiedenen Detailgraden, wie oben beschrieben, gewählt, sollten die betroffenen Elemente sanft ein- und ausgeblendet werden.

Aufschlüsselung der Symbole und Farben Zentraler Punkt einer Heatmap ist die Darstellung eines Wertes in einer bestimmten Farbe, im gegebenen Fall die Auslastung des logischen Prozessors oder der Ressourcenverbrauch eines Threads. Dadurch ist die Bedeutung der Farbe eines Icons bereits festgelegt. Es besteht also keine Möglichkeit, die Iconfarbe zur Darstellung anderer Informationen, wie im Prototypen geschehen, zu nutzen.

Fazit Die in einer Heatmap eingesetzten Farbcodierungen und Kontraste können vom Anwender schneller erkannt und unterschieden werden als reiner Text. Daher eignen Heatmaps sich insbesondere, um einen schnellen Überblick über eine große Menge von Werten zu gewinnen, die fest mit einem Ursprung verknüpft sind. Beispiele wären die Verknüpfung einer Temperatur mit einem Punkt im Raum oder, im Rahmen dieser Arbeit, die Auslastung eines Prozessors mit diesem Prozessor.

Die Übersichtlichkeit wird jedoch hauptsächlich durch Datensparsamkeit erreicht - wird mehr als eine Eigenschaft eines Objekts durch Farbkodierung dargestellt, verliert sich der Vorteil schnell, da nun zusätzlich eine Verbindung zwischen Farbe und Eigenschaft hergestellt werden muss. Da im Verlauf eines Programmablaufs sehr viele Threads gestartet werden können, kann hier schnell die Übersicht verloren gehen.

3.5 3D Time Wheel

Ein 3D Time Wheel ist eine von Tominski et al. [2005] vorgestellte Visualisierungstechnik. Es ist eine Erweiterung eines zweidimensionalen Time Wheels. Zusätzliche Navigationsmöglichkeiten und Perspektiven im dreidimensionalen Raum bieten hier Vorteile bezüglich der Übersichtlichkeit gegenüber einer zweidimensonalen Darstellung. Ein (3D-) Time Wheel kann eingesetzt werden, um zeitabhängige, multivariate Daten anzuzeigen. Das bedeutet, jeder angezeigte Attributwert (z.B. ein Messwert) eines Datensatzes kann einem eindeutigen Zeitstempel zugeordnet werden.

Als Ausgangspunkt dient eine zentrale *Zeitachse* (siehe Abb. 5). Parallel zu dieser Zeitachse sind *Attributachsen* angeordnet. Die Anzahl der Attributachsen entspricht der Anzahl der zeitabhängigen Attribute eines Datensatzes. Auf den Attributachsen sind die Wertebereiche der zeitabhängigen Attribute aufgetragen. Dabei müssen weder die Wertebereiche der Zeitachse noch der Attributachsen linear sein; Tominski et al. [2005] stellen in ihrem Ansatz auch die Möglichkeit vor, den aufgetragenen Wertebereich zu verändern, beispielsweise durch Stauchen oder Verschieben.

Zur Darstellung der Verbindung eines Attributwerts mit einem bestimmten Zeitpunkt wird eine Linie von dem Wertebereich Attributs (also seiner Attributachse) aus zu dem entsprechenden Punkt auf der Zeitachse gezogen. Dadurch kann der Anwender die Verbindung zwischen Attributwert und Zeitstempel wiederherstellen.

In derselben Arbeit von Tominski et al. [2005] werden Varianten des 3D Time Wheel vorgestellt, die 3D Kiviat Tube und der 3D Multicomb. Beide sind für ähnliche Einsatzbereiche wie das Time Wheel gedacht.

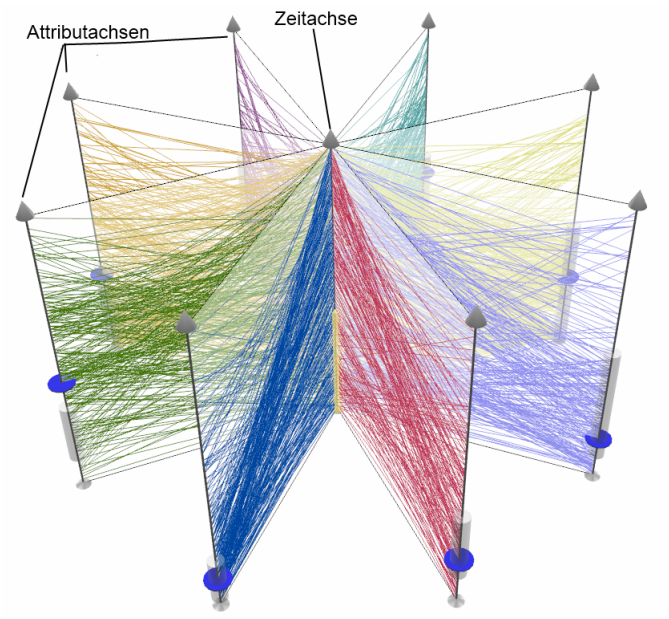


Abbildung 5: Time Wheel mit 8 Attributen. Quelle: Tominski et al. [2005], Beschriftungen hinzugefügt

Ansatz im Rahmen der Prozessorauslastung

Um ein 3D Time Wheel in unserem Szenario einsetzen zu können, muss zunächst die Entscheidung getroffen werden, welche Zeitachse verwendet wird. Zum einen besteht die Möglichkeit, die Zeitachse eines logischen Prozessors zu verwenden und für jeden logischen Prozessor ein eigenes Time Wheel zu erstellen. In diesem Fall wäre der augenblicklich ausgeführte Thread ein Wert auf einer Attributachse, deren Wertebereich aus der Menge der Threads besteht.

Zum anderen ist es möglich, die Zeitachse eines Threads zu verwenden und ein Time Wheel Thread darzustellen. Damit würde der Wertebereich einer Attibutachse mit der Menge der zur Verfügung stehenden logischen Prozessoren gefüllt.

Die Wahl der Zeitachse bestimmt dabei den Fokus der Visualisierung. Die erste Möglichkeit sollte gewählt werden, wenn das Interesse des Benutzers der Auslastung des logischen Prozessors gilt. Dies ist der Fall, wenn ein komplettes System analysiert werden soll, in dem die momentan auf einem logischen Prozessor ausgeführten Threads nur als Ursache für die bestehende Prozessorlast betrachtet werden.

Die zweite Möglichkeit beleuchtet das Verhalten der Threads genauer. Sie kommt zum Einsatz wenn, der Benutzer eine feiner aufgelöste Darstellung (auf Methodenebene) der Aktivitäten eines oder mehrerer Threads benötigt. In dieser Darstellung verliert sich die

Übersicht über das Gesamtsystem.

Der vorgestellte Ansatz wählt die Zeitachse des logischen Prozessors als zentrale Achse, da dies dem Ansatz der Arbeit näher kommt.

Vollständigkeit Ein 3D Time Wheel für das gewählte Szenario benötigt 3 Attributsachsen. Zuerst würde die Auslastung des logischen Prozessors hinzugefügt. Da der Wertebereich einer Attributachse nicht auf numerische Werte beschränkt ist, kann als weitere Attributachse die möglichen aufgerufenen Methoden genommen werden. Als nächstes würde die Menge der möglichen Threads hinzugefügt. Damit ist ein Time Wheel in der Lage, die Kerndaten, nach der harten Definition der Vollständigkeit, vollständig anzuzeigen.

Granularität Die Granularität der Visualisierung hängt, wie oben bereits beschrieben, hauptsächlich von der Wahl der Zeitachse ab. Die Möglichkeit, die Skalierung der Zeitachse und der Attributachsen zu verändern, bietet dem Anwender die Freiheit, die Granularität selbst zu bestimmen. Durch das Hinzufügen oder Entfernen von Attributachsen kann die Granularität der Visualisierung weiter beeinflusst werden. Besonders das eingrenzen der Zeitachse auf einen bestimmten Zeitraum erlaubt sowohl eine gute Detailansicht sowie einen weiten Überblick über alle Daten.

Räumliche Aufteilung und Orientierung Das 3D Time Wheel ist einfach aufgebaut. Durch Beschriftung der Attributachsen kann eine leichte Orientierung ermöglicht werden. Da die Zeitachse stets in der Mitte der Attributachsen dargestellt wird, bietet sie einen Orientierungspunkt, wenn der Blickwinkel des Anwenders verschoben wird, beispielsweise durch Rotation oder Verschieben der Kamera. Wenn gut zu unterscheidende Farben für die Linien zwischen Zeit- und Attributachse gewählt werden, können diese als weitere Orientierung dienen.

Die Anordnung der Attributachsen im Kreis um die Zeitachse kann sich als nachteilig erweisen, wenn ihre Anzahl steigt. Eine Möglichkeit ist, die Attributachsen enger zusammenzurücken. Stehen die Achsen jedoch zu eng zusammen, können evtl. die Linien zur Zeitachse nicht mehr klar unterschieden werden. Eine zweite Möglichkeit ist, die Attributachsen weiter von der Zeitachse zu entfernen. Wandern Attribut- und Zeitachse jedoch zu weit auseinander, erschwert dies die Zuordnung von Wert und Zeitpunkt.

Animation In einer 3D Time Wheel Visualisierung ist das Schrittweise aufbauen der Darstellung entlang der Zeitachse denkbar. Wenn die Anzahl oder die Position der Attributachsen verändert werden, bietet sich eine Animation der Zustandsänderung an, um die Orientierung zu erleichtern.

Aufschlüsselung der Symbole und Farben Das Hauptaugenmerk beider Aufschlüsselung der Symbole und Farben liegt auf den Verbindungslinien zwischen Zeit- und Attributachse. Die farbliche Markierung von Attributen erlaubt eine schnelle Orientierung innerhalb der Visualisierung. Zur weiteren Orientierung und Eingrenzung könnte ein Schiebregler auf der Zeitachse hinzugefügt werden. Mit diesem Schiebregler könnte ein Bereich ausgewählt werden, in dem alle momentanen Verknüpfungen in einer Signalfarbe

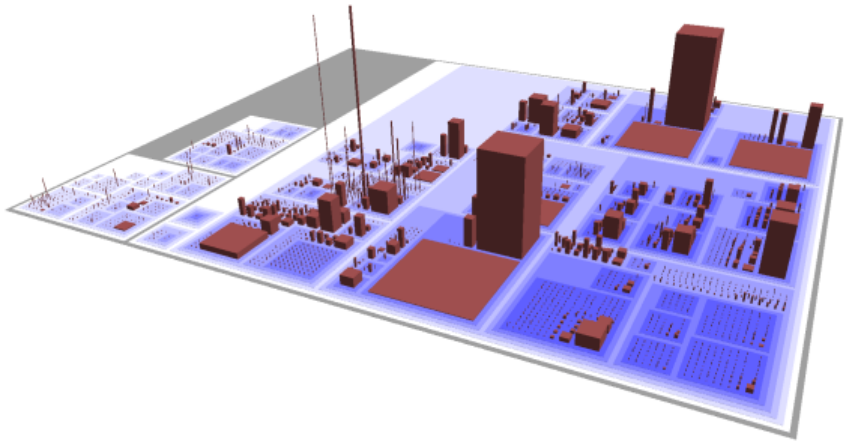


Abbildung 6: Visualisierung einer Softwarestruktur mit CodeCity. Quelle Wettel and Lanza [2008]

dargestellt werden. Alternativ könnte statt einer einheitlichen Signalfarbe ein anderer Farbton der Ursprungsfarbe verwendet werden.

Fazit In einem Time Wheel sind der Startzeitpunkt einer Methode und der Thread, auf dem diese Methode ausgeführt wird, direkt miteinander verknüpft. Dadurch kann die Methode, die auf diesem Thread am häufigsten ausgeführt wurde, leicht erkannt werden. Auch allgemeinere zeitliche Zusammenhänge können leicht erkannt werden. Dies ist in den anderen beschriebenen Visualisierungen nur durch eine zusätzliche Anzeige möglich. Darin hat das (3D-)Time Wheel einen Vorsprung in der „semantischen Vollständigkeit“ (siehe Kap. 2.2.1).

Größter Nachteil dieser Visualisierungstechnik ist die zuvor zu treffende Wahl des Fokus der Analyse, da ein Umstellen der Daten im Zweifelsfalle nicht trivial ist und ein Neuaufbereiten erfordert. Zudem engt es die Möglichkeit ein, Beziehungen zwischen Thread und logischem Prozessor zu erkennen. Damit verliert es den gewonnenen Vorsprung wieder.

Dennoch ist ein Time Wheel eine adäquates Mittel zur Visualisierung von Prozessorauslastungen, wenn der gewählte Fokus mit dem Ziel der Analyse übereinstimmt.

3.6 City Metapher

Eine Visualisierung auf Basis einer *City Metapher* stellt große hierarchische Strukturen als Stadt dar. Dabei wird jedem Objekt der Stadt (Haus, Stadtteil...) ein Objekt der Struktur (Paket, Methode, Klasse...) zugeordnet. Jede Eigenschaft eines abstrakten Objektes wird durch eine Eigenschaft des Objekts der Stadt dargestellt. Die Höhe eines Hauses, das eine Java-Klasse darstellt, könnte beispielsweise von der Anzahl der Codezeilen der Klasse abhängen. Der Gedanke dahinter ist, eine abstrakte Struktur in eine für

Menschen vertraute Form zu bringen, in der sie navigieren und sich orientieren können. Dadurch soll das Begreifen des Systems vereinfacht werden [Wettel and Lanza 2007]. City Metaphern finden sich beispielsweise, wie oben angedeutet, in der Visualisierung von großen Softwarestrukturen (siehe Abb. 6 oder [Alam and Dugerdil 2007]) oder auch in der Darstellung von Dateisystemen (z.B. „The Deleted City“ [Vijgen 2009]).

Ansatz im Rahmen der Prozessorauslastung

Die *City Metapher* bietet einen einfachen Weg, die zentralen Daten, symbolisch und übersichtlich darzustellen. Daher ist das Design des Prototypen angelehnt an die City Metapher. Der Prototyp ist eine einfache Variante der City Metapher, die nur die Farbe und die Höhe der Gebäude sowie die Zusammenfassung von Häusern zu Stadtteilen mit einer Bedeutung versieht. Dabei werden beide Werte, aktuelle Laufzeit und Name des Threads, zusätzlich auch textuell dargestellt. Für die restlichen der Visualisierung zur Verfügung stehenden Daten, z.B. *Hungriest Method*, konnte keine zur City Metapher passende symbolische Darstellung gefunden werden, weswegen sie nur rein textuell dargestellt werden.

Eine Evaluierung des verwendeten Ansatzes auf Basis der City Metapher wird in Kapitel 6 genauer behandelt. An dieser Stelle werden einige allgemeine Eigenschaften der City Metapher diskutiert.

Vollständigkeit Durch die Möglichkeit, jede Eigenschaft eines Gebäudes oder einer Stadt als Repräsentanten für eine Eigenschaft des abstrakten Objektes einzusetzen, ist eine City Metapher ein ausdrucksstarkes Werkzeug. Neben Eigenschaften wie Höhe, Breite und Länge eines Gebäudes kann auch das Aussehen des Hauses selbst als Repräsentant genutzt werden. Beispielsweise könnte der Zustand des Gebäudes (neu, gepflegt, heruntergekommen, verfallen..) den Zeitpunkt des letzten Zugriffs auf eine Datei darstellen oder die Anzahl der Fenster könnte Aufschluss über die Dateifreigaben geben.

Begrenzungen liegen hier in der Verständlichkeit für den Anwender. Um eine City Metapher effizient nutzen zu können, muss er alle für ihn wichtigen Eigenschaften erkennen können. Eine zu hohe Anzahl von Repräsentanten ist hierbei weniger gefährlich als eine willkürliche oder unverständliche Auswahl. Die Größe einer Datei mit der Höhe eines Gebäudes zu verbinden fällt dem Anwender leicht. Die Höhe mit der Zeitspanne, die seit der letzten Aktualisierung der Datei vergangen ist (je höher, desto größer die Zeitspanne), zu assoziieren ist hingegen eher unnatürlich.

Granularität Zoom- und Navigationsfunktionen sind für umfangreiche Städte essentiell, ebenso wie die Möglichkeit, ohne Zeitverlust zu einem anderen Stadtteil springen zu können. Auch das Ausblenden von Stadtteilen (analog zum einklappen eins Verzeichnisbaums) sollte vorgesehen werden.

Räumliche Aufteilung und Orientierung Die räumliche Aufteilung ist die Paradedisziplin der City Metapher. Sie ist speziell darauf ausgerichtet, dem Anwender eine für ihn „natürlich“ anmutende Umgebung zu schaffen und ihm so die Orientierung zu erleichtern.

Animation Auch Animation kann innerhalb einer City Metapher als Repräsentant genommen werden. So könnte beispielsweise die Häufigkeit des Dateiaustausches zwischen zwei (Netz-)Laufwerken über eine animierte Straße mit unterschiedlich hohem Verkehrsaufkommen dargestellt werden. Die Geschwindigkeit der Fahrzeuge würde dann die Lese-/Schreibgeschwindigkeit anzeigen.
Aufschlüsselung der Symbole und Farben Wie bereits oben angemerkt kann die größte Stärke der City Metapher, ihre große Anzahl an Symbolen, auch in ein Problem umschlagen, wenn nicht entsprechend vorsichtig entworfen wurde. Generell ist eine Aufschlüsselung aller Symbole in Form einer Bedienungsanleitung Pflicht bei der Verwendung einer großen Anzahl an Repräsentanten.
Fazit City Metaphern bieten unzählige Möglichkeiten zur Darstellung von statischen Systemen. Jedoch ist die Darstellung von sich zur Laufzeit verändernden Daten, wie sie bei der Visualisierung von Prozessorauslastung vorliegen, mit einem gewissen Implementierungsaufwand verbunden. Zusätzlich besteht das schwerwiegende Risiko, dass der Anwender in einer sich ständig verändernden Stadt die Übersicht verliert. Näheres dazu findet sich in Kapitel 6.

3.7 ParaGraph

Im Gegensatz zu den bisher beschriebenen Visualisierungen ist *ParaGraph* keine einzelne Visualisierungstechnik. ParaGraph (von Heath and Finger [2003]) ist ein Werkzeug, mit dem das Verhalten eines Mehrprozessorsystems analysiert werden kann. Es verfolgt damit ein ähnliches Ziel wie der im Rahmen dieser Arbeit erstellte Prototyp. Dazu verwendet es ca. 30 verschiedene Visualisierungstechniken (sogenannte *Displays*), die alle aus den gleichen Daten gespeist werden [Heath and Finger 2003]. Die Displays sind dabei in die Kategorien *Utilization*, *Communication* und *Task* unterteilt.
Utilizationdisplays sind dazu gedacht, den Status der Prozessoren darzustellen. Die ermittelbaren Zustände sind *busy*, *idle* und *overhead*. *Busy* bedeutet, das der Prozessor zur Zeit mit der Abarbeitung der eigentlichen Anwendung beschäftigt ist. *Idle* zeigt an, das der Prozessor zur Zeit nicht benutzt wird. *Overhead* steht dafür, dass der Prozessor gerade mit der Verwaltung seiner Prozesse beschäftigt ist. Utilization Displays ähneln in ihrer Zielsetzung der vorgestellten Arbeit am stärksten. Sie bieten jedoch keine Zuordnung von Prozessen zu Prozessoren.
Communicationdisplays stellen die Kommunikation der Prozesse untereinander dar. Mit ihnen ist es möglich, beispielsweise die Länge von Message Queues oder die Menge an ausgetauschten Nachrichten darzustellen.
Taskdisplays bieten dem Anwender die Möglichkeit, eigene Daten erfassen zu lassen. Ein Task ist dabei eine vom Anwender wählbare Aufgabe innerhalb der Anwendung, beispielsweise die Berechnung eines bestimmten, komplexen Wertes. Jeder Task bekommt eine eindeutige Nummer, über die er in der Visualisierung identifiziert werden kann. Dazu muss jedoch der Quellcode der Anwendung modifiziert werden. Beginn und Ende

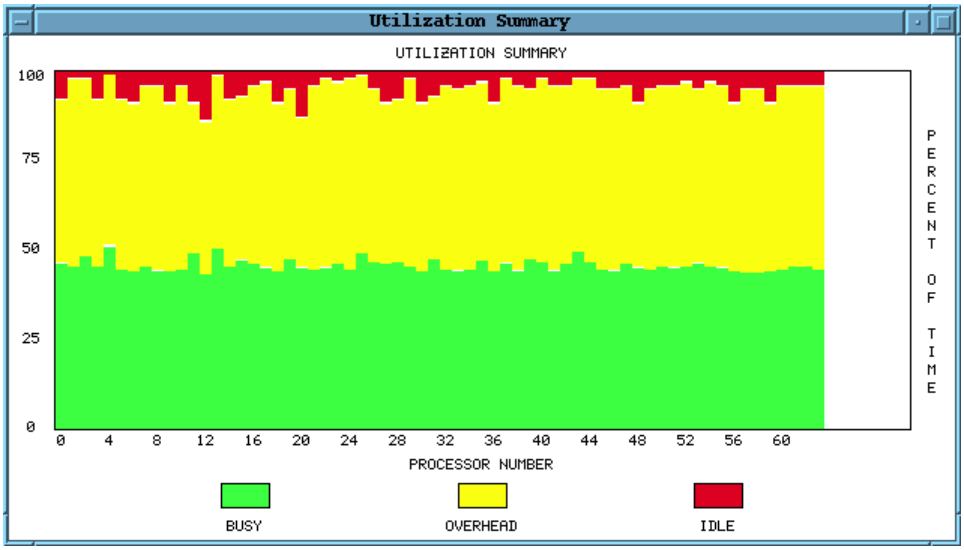


Abbildung 7: Utilization Summary Display von ParaGraph. Quelle: [Lauxtermann 1995]

eines Tasks werden innerhalb des Quellcodes durch den Aufruf einer speziellen Funktion markiert.

Eine weitere Kategorie, *Other*, umfasst Displays, die nicht in die drei anderen Kategorien passen.

ParaGraph bedient sich zur Datenerfassung des *Message Passing Interfaces (MPI)*. Dadurch kann ParaGraph Anwendungen analysieren, die aus mehreren, per MPI kommunizierenden, Prozessen bestehen. Eine Möglichkeit, die Kommunikation auf Threadebene zu verfolgen, besteht nicht. Tasks benötigen zusätzlich MPICL, eine Bibliothek zur Erfassung von Daten innerhalb von MPI Anwendungen (siehe Worley [2005]).

Die jüngsten auffindbaren Informationen zu ParaGraph stammen jedoch aus dem Jahr 2003, was auf eine Einstellung des Projekts hinweist. Daher ist ParaGraph technisch und graphisch heutzutage nicht mehr auf einem aktuellen Stand. Abbildung 7 zeigt das *Utilization Summary* Display von ParaGraph, das eine Zusammenfassung eines Simulationsverlaufs zeigt.

Sowohl der Prototyp und ParaGraph sind Werkzeuge zur Optimierung von Multiprozessorsystemen, jedoch unterscheiden sie sich stark in Ansatz und Ausführung. Während ParaGraph beim Nachrichtenaustausch zwischen zwei laufenden Prozessen ansetzt, arbeitet der Prototyp auf Methodenebene, kann also auch Daten sammeln, die innerhalb der Anwendung anfallen. ParaGraph unterstützt dies nur, wenn der zu untersuchende Quellcode händisch angepasst wird. Der Prototyp arbeitet mit aspektorientierter Programmierung, kann also seine Daten ohne Veränderung des Quellcodes der Zielanwen-

dung sammeln.

Während der Prototyp Daten auf Threadebene sammelt, arbeitet ParaGraph auf Prozessebene. Dadurch kann der Prototyp eine feiner aufgelöste Ansicht des aktuellen Systemzustands liefern. Zusätzlich kann der Prototyp Thread zu dem logischen Prozessor zuordnen, der ihn momentan ausführt.

Auch der Ansatz zur Darstellung der Daten unterscheidet sich der Prototyp von ParaGraph. Während der Prototyp alle Informationen in einer einzelnen 3D-Ansicht darstellt, setzt ParaGraph auf eine Vielzahl von 2D-Ansichten. Jede der 2D-Ansichten erlaubt es dem Anwender, einen anderen Aspekt genauer zu untersuchen.

Anforderung	Zugehörige Kriterien
Datenerfassung und angezeigte Daten	Vollständigkeit
Visuelles Design	Vollständigkeit, Animation, Räumliche Aufteilung / Orientierung, Aufschlüsselung der Informationen
Kamera und Bedienung	Granularität, Räumliche Aufteilung / Orientierung
Implementierung und Architektur	Vollständigkeit

Tabelle 2: Zuordnung von erkannten Anforderungen zu den verwendeten Kriterien

4 Konzept

Basierend auf den in Kapitel 2.2 vorgestellten Kriterien *Vollständigkeit*, *Granularität*, *Räumliche Aufteilung / Orientierung*, *Animation* und *Aufschlüsselung der Informationen* wurden Anforderungen an den zu implementierenden Prototypen erstellt. Zusätzlich zu den genannten Kriterien wurde bei der Erstellung der Anforderungen das Einsatzgebiet, die Überwachung der Kernauslastung in einem Multiprozessorsystem, berücksichtigt. Dabei ergaben sich verschiedene Bereiche, in die sich die Anforderungen unterteilen lassen: *Visuelles Design*, *Kamera und Bedienung*, *Datenerfassung und abgeleitete Informationen* und *Implementierung und Architektur*. Jeder der genannten Bereiche kann weitere Unterbereiche enthalten.

Tabelle 2 zeigt, aus welchen Kriterien sich die einzelnen Anforderungen ableiten.

4.1 Visuelles Design

Gemäß des Themas der Arbeit wurden zuerst die Anforderungen an das visuelle Design des Prototypen definiert. Dabei ergibt sich in diesem Bereich die Frage, welcher der in Kapitel 3 genannten Visualisierungen der Prototyp zugrunde liegen soll. Weiter ergibt sich die Frage nach der Art und Weise, wie der zeitliche Verlauf der Daten dargestellt werden wird.

4.1.1 City Metapher

Die City Metapher bietet eine hohe Flexibilität in Bezug auf die Darstellung von Informationen, zusätzlich zu einer hohen Übersichtlichkeit. Daher wird sich der Prototyp visuell an der City Metapher (siehe Kapitel 3.6) orientieren.

Abbildung 8 zeigt einen Entwurf des Prototypen, der sich an der City Metapher orientiert. Der Entwurf ist so gestaltet, dass die Hierarchie eines Multiprozessorsystems für den Anwender erkennbar ist. Auf dem Entwurf sind die einzelnen Threads in Form

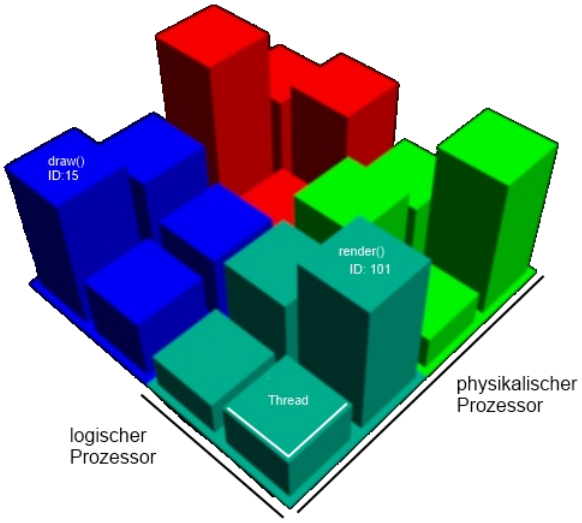


Abbildung 8: Konzeptentwurf des Prototypen

von Häusern zu sehen. Die Häuser sind zu Stadtteilen oder Blocks zusammengefasst, die einen einzelnen logischen Prozessor repräsentieren. Dabei beinhaltet jeder Häuserblock ein Haus für jeden Thread, der im System bekannt ist. Mehrere logische Prozessoren werden zu einer physikalischen CPU zusammengefasst. Eine komplette Stadt ergibt sich dann aus mehreren physikalischen CPUs. Das Layout der einzelnen Parts ist möglichst einfach gehalten, um Übersichtlichkeit zu gewähren. Dazu bieten sich Rechteckformen an. Durch ein klares Layout und die deutliche Unterscheidung der einzelnen Objekte wird das Kriterium *Räumliche Aufteilung / Orientierung* (Kapitel 2.2.3) erfüllt.

Als nächstes wurde überlegt, welche Informationen innerhalb der Stadt dargestellt werden sollen. Dabei ergeben sich der *Name des ausführenden Threads*, der *Name der ausgeführten Methode*, die *verbrauchte Zeit* und die *CPU Id* direkt aus dem Kriterium *Vollständigkeit*. Während der Erstellung der Anforderungen wurde zudem festgestellt, dass die *Hungriest Method*, also die Methode, die die höchste Anzahl an Aufrufen auf diesem Thread besitzt, für einen Anwender nützlich ist. Durch die *Hungriest Method* kann der Anwender feststellen, welche Methoden eventuell sehr oft aufgerufen werden, also optimiert werden sollten. Daher wird der Prototyp zusätzlich die *Hungriest Method* anzeigen können.

Die genannten Informationen werden wie folgt dargestellt:

Name des ausführenden Threads Bei Namen handelt es sich von sich aus um textuelle Informationen, die am Besten durch eine textuelle Darstellung repräsentiert werden. Um

den Namen unabhängig von der aktuellen Kameraposition sehen zu können, wird er auf dem Dach des Hauses abgebildet, das diesen Thread repräsentiert.

Zusätzlich soll ein Thread innerhalb eines Blocks leicht wiederzufinden sein. Daher wird ein Thread mit einer bestimmten Id stets denselben Platz in jedem Block einnehmen.

Name der ausgeführten Methode Für den Namen der ausgeführten Methode gelten die gleichen Bedingungen wie für den Namen des ausgeführten Threads. Daher wird auch der Methodenname in textueller Form auf dem Dach abgebildet.

Verbrauchte Zeit Die Zeit, die ein bestimmter Thread innerhalb eines Intervalls auf einem logischen Prozessor zugebracht hat, wird durch die Höhe des ihm zugeordneten Hauses dargestellt. Dadurch ist ein schneller Überblick über die verbrauchten Zeiten möglich. Da die Höhe der Häuser nur eine relative Aussage zulässt („viel“ und „wenig“), nicht aber eine genaue Angabe („26%“), wird der genaue Prozentwert zusätzlich in textueller Form auf dem Dach angezeigt.

CPU Id Die Anzeige der CPU Id eines logischen Prozessors wird für jeden Block in textueller Form erfolgen. Durch die räumliche Zusammenfassung der Häuser zu Blocks ist damit eine einfache Zuordnung von beobachtetem Thread zu ausführendem logischen Prozessor möglich.

Hungriest Method Die *Hungriest Method* wird, analog zum Namen der ausgeführten Methode, auf dem Dach des Hauses in textueller Form angezeigt. Dabei muss sichergestellt werden, das die beiden Methodennamen voneinander unterscheidbar sind.

Diese Anforderungen ergeben sich aus dem Kriterium *Aufschlüsselung der Informationen* (Kapitel 2.2.5).

4.1.2 Animation

Der Prototyp wird den Verlauf einer Messreihe animiert wiedergeben können. Dazu werden die Übergänge von einem Messpunkt zum nächsten so animiert, dass die Identifizierung von Objekten sowohl vor als auch nach der Durchführung eines Schrittes möglich ist.

Diese Anforderungen ergeben sich aus den Kriterien *Räumliche Aufteilung / Orientierung* (Kapitel 2.2.3) und *Animation* (Kapitel 2.2.4).

4.2 Kamera und Bedienung

Die Benutzbarkeit des Prototypen hängt stark von den Möglichkeiten des Benutzers zur Manipulation der Darstellung ab. Daher werden diese beiden Punkte gemeinsam aufgeführt. Dabei werden die Anforderungen aus den Kriterien *Räumliche Aufteilung / Orientierung* (Kapitel 2.2.3) und *Granularität* (Kapitel 2.2.2) abgeleitet. Um eine einfache und übersichtliche Darstellung zu erhalten, werden Anforderungen an die verfügbaren Kamerapositionen und die Steuerung des Prototypen gestellt.

4.2.1 Kamerapositionen

Der Benutzer muss in der Lage sein, die Kamera frei nach seinen Wünschen auszurichten. Dadurch kann er den für ihn interessanten Ausschnitt der Darstellung wählen. Dies ist wichtig, um eine höhere *Granularität* des Prototypen zu erhalten. Zusätzlich muss die Möglichkeit gegeben sein, feste Punkte anzuspringen. Diese festen Punkte sollen dem Benutzer eine Orientierungshilfe geben, zu der er jederzeit zurückkehren kann. Damit tragen sie zur Erfüllung des Kriteriums *Räumliche Aufteilung / Orientierung* bei. Zudem sollen sie neuen Benutzern einen Anfangspunkt liefern, an dem sie einen Überblick über die angezeigten Daten bekommen.

4.2.2 Steuerung

Um dem Anwender die Bedienung des Prototypen zu erleichtern und so das Kriterium *Räumliche Aufteilung / Orientierung* möglichst gut zu erfüllen, wird die Steuerung der Kamera so einfach wie möglich gehalten werden. Die freie Ausrichtung der Kamera wird mindestens über die Tastatur möglich sein. Um die Kamera an jedem Punkt der Darstellung positionieren zu können, muss die Kontrolle der Kameraposition verschiedene Bewegungsrichtungen umfassen. Es müssen mindestens die Richtungen *Vor* und *Zurück* zur Verfügung stehen, zusätzlich zu der Möglichkeit, die Kamera zur Seite, nach Oben und nach Unten zu schwenken.

Hilfreich, aber nicht notwendig, ist eine Erweiterung der Steuerung um eine Maussteuerung, die die Kameraschwenks übernimmt und einer seitlichen Bewegung der Kamera über die Tastatur. Diese Form der Navigation innerhalb von 3D-Umgebungen ist besonders aus dem Spieleumfeld bekannt und hat sich dort bewährt [Wiki2011]. Optimal für eine einfache Bedienung wäre eine frei konfigurierbare Steuerung. Diese beiden Punkte sind für eine grundlegende Funktion des Prototypen optional. Der Prototyp wird sich daher auf eine festgelegte Tastatursteuerung beschränken.

4.3 Datenerfassung und abgeleitete Informationen

Der Prototyp muss in der Lage sein, die dargestellten und zur Darstellung benötigten Daten aus dem System auszulesen. Die benötigten Daten sind die Id des momentanen logischen Prozessors, der Name des momentanen Threads, der Name der aktuellen Methode sowie ein aktueller Zeitstempel.

Bei der Erfassung der notwendigen Daten soll so weit wie möglich Plattformunabhängigkeit gewahrt werden. Durch die Erfassung und Darstellung der genannten Daten kann der Prototyp das Kriterium der *Vollständigkeit* (siehe Kapitel 2.2.1) erfüllen.

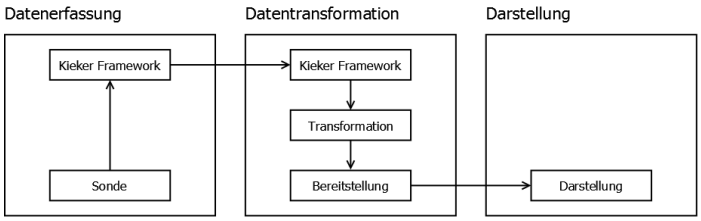


Abbildung 9: Ansatz der Architektur des Prototypen

4.4 Implementierung und Architektur

Der Prototyp wird modular gestaltet werden, so dass einzelne Teile ersetzt werden können. Die identifizierbaren Einzelteile sind Datenerfassung, Datentransformation und Darstellung. Die Architektur des Prototypen wird diese Unterteilung widerspiegeln. Abbildung 9 zeigt die geplante Struktur der Implementierung. Insbesondere muss die Datenerfassung eigenständig lauffähig sein, um Overhead durch die anderen Parts zu vermeiden. Oftmals sind zu analysierende Systeme unterschiedlich in Bezug auf Prozessorarchitektur und Betriebssystem. Zusätzlich kann sich auch das System, auf dem die gesammelten Daten ausgewertet werden, von dem analysierten System unterscheiden. Dies ist beispielsweise der Fall, wenn ein Produktivsystem untersucht werden soll, das speziell für eine Aufgabe erstellt wurde. In diesem Fall muss die Auswertung auf einem unabhängigen System stattfinden, um den normalen Betriebsablauf nicht zu stören. Daher muss auch der durch die Datenerfassung generierte Overhead möglichst gering gehalten werden. Eine eigenständige Datenerfassung kann leichter auf verschiedene Plattformen portiert werden, ohne die Datentransformation oder die Darstellung zusätzlich anpassen zu müssen.

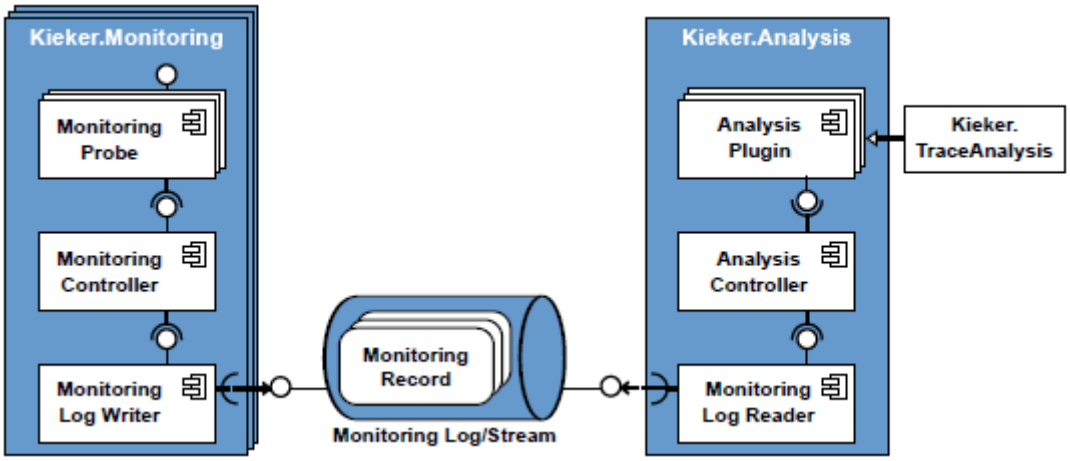


Abbildung 10: Kieker Framework. Quelle: Kieker Project [2012b]

5 Implementierung

In den vorangegangenen Kapiteln wurden die theoretischen Grundlagen und eine Auswahl bereits bekannter Visualisierungstechniken beschrieben. Es wurden Vorschläge beschrieben, wie mit den bestehenden Visualisierungstechniken Prozessorauslastungen dargestellt werden können.

Das folgende Kapitel beschäftigt sich mit der Implementierung eines Prototypen auf Grundlage einer vereinfachten City Metapher (siehe Kapitel 3.6). Dazu werden zuerst die dafür verwendeten Werkzeuge beleuchtet, anschließend werden Aufbau und Details der Implementierung beschrieben. Der Prototyp ist in sich ein Vorschlag, wie eine Visualisierung von Prozessorauslastungen aussehen kann.

5.1 Werkzeuge / Technologien

Bevor genauer auf den Prototypen eingegangen wird, werden in diesem Abschnitt die Werkzeuge und Bibliotheken beschrieben, die zur Implementierung benötigt wurden.

5.1.1 Kieker

Das Kieker Framework [Kieker Project 2012a] ist ein an der Kieler Universität entwickeltes Werkzeug zum Monitoring und zur Analyse von (Java-) Software. Der Fokus der Analyse liegt dabei auf dem *Monitoring*, also dem Überwachen und Protokollieren des Laufzeitverhaltens einer Anwendung. Abbildung 10 zeigt einen Überblick über die Struktur des Kieker Frameworks. Dazu verwendet Kieker sogenannte Sonden (auch *Probes*),

die auf verschiedene Weisen in eine Anwendung eingebettet werden können, um dort zur Laufzeit Daten zu sammeln. Die gesammelten Daten werden von der Sonde an einen *Monitoring Controller* weitergeleitet. Der Monitoring Controller verwaltet mehrere eingebettete Sonden und bündelt ihre Ausgabe. Die Ausgaben werden an einen oder mehrere *Monitoring Writer* weitergeleitet, die die Ausgaben in einer vereinheitlichten Form von *Monitoring Records* ausgeben. An dieser Stelle erfolgt durch die Monitoring Records eine Trennung zwischen dem Daten sammelnden Part des Kieker Frameworks und dem zweiten Part, der Analyse. Der Analyse Part nimmt die vom Monitoring Part erzeugten Monitoring Records mithilfe eines *Monitoring Readers* entgegen. Monitor Reader und Writer können dabei entweder synchron oder asynchron arbeiten. Ein synchroner Datenaustausch wäre beispielsweise für das Debuggen einer Anwendung in Echtzeit notwendig. Eine asynchrone Übertragung, bei der die Monitoring Records in das Dateisystem oder in eine Datenbank geschrieben werden, erlaubt es, die Daten vor der eigentlichen Analyse oder Visualisierung aufzubereiten.

Der Prototyp setzt auf das Kieker Framework auf. Der Prototyp nutzt die vom Framework zur Verfügung gestellten Mechanismen zum sammeln, speichern und auslesen von Messdaten. Dafür wurde eine eigene Sonde zur Datenerfassung implementiert (siehe Kapitel 5.4).

Kieker bietet eine bestehende Infrastruktur für das Monitoring und die Analyse von Softwaresystemen. Es ist durch bestehende Interfaces komfortabel erweiterbar. Der Overhead durch die Datenerfassung wird im Kieker Framework gezielt niedrig gehalten (siehe [van Hoorn et al. 2012]).

5.1.2 CPUID

Um eine Zuordnung von einem Thread zu dem logischen Prozessor, auf dem er ausgeführt wird, vornehmen zu können, benötigt die Kieker Sonde einen Mechanismus, der Zugriff auf die ID des logischen Prozessors erlaubt. CPUID ist eine Assemblerinstruktion zum Auslesen prozessorspezifischer Daten, die für diesen Zweck geeignet ist.

Mit CPUID können Daten wie Hersteller, Modell und Familie ausgelesen werden. Speziell können auch Informationen über die Architektur, also Anzahl und IDs der logischen Prozessoren, maximale Threadzahl und ähnliches ausgelesen werden. Jeder aktuelle x86-kompatible Prozessor beherrscht mindestens einen standardisierten und herstellerübergreifenden Satz an CPUID Funktionen. Beispielsweise beherrschen Intel Prozessoren die CPUID Funktion bereits seit der Intel486 Generation. Zusätzlich beherrschen Prozessoren einzelner Hersteller meist noch herstellerspezifische Funktionen.

Konkret findet CPUID im Daten sammelnden Part des Prototypen Anwendung. Der Aufruf von von CPUID wird in eine *Kieker*-Sonde eingebettet, die in einer Methode platziert wird (vgl. Kap. 5.4). Wird die Datenerfassung der Sonde ausgelöst, ermittelt sie durch CPUID die ID des logischen Prozessors, der die gemessene Methode ausführt. Gleichzeitig kann die von der Methode verbrauchte Zeit erfasst werden. Dadurch kann zu

jedem Messzeitpunkt eine näherungsweise Zuordnung der tatsächlich verbrauchten Zeit eines Threads (die sog. *Wall Time*) zu dem logischen Prozessor, auf dem die Zeit verbraucht wurde, erstellt werden. Diese Zuordnung wird im Visualisierungsteil ausgewertet und dargestellt.

Im Vorfeld einer Datenerfassung wird zudem eine Karte des Systems erstellt, also die Anzahl der physikalischen Prozessoren und Anzahl der logischen Prozessoren pro physikalischem Prozessor. Zudem wird erfasst, welcher logische Prozessor welchem physikalischen Prozessor zugeordnet ist. Da diese Daten für den gesamten Lauf der Datenerfassung gleich bleiben, braucht in der Visualisierung nur die Zuordnung der Threads für jeden Messzeitpunkt neu ermittelt dargestellt werden.

CPUID wird aufgerufen, indem die gleichnamige Assemblerfunktion aufgerufen wird. Zuvor wird ein Parameter in das Register EAX geschrieben, der Anzeigt, welche Informationen benötigt werden. Daraus ergeben sich zur Identifikation einzelner CPUID Aufrufe Schreibkonventionen wie *CPUID_Fn0000_0000_EAX* [AMD]. Dieser Befehl liest sich wie folgt:

- CPUID Die Assemblerinstruktion, in diesem Beispiel CPUID.
- Fn0000_0000 Das zu setzende Bitmuster in Hexadezimalschreibweise, in diesem Falle *0000* sowohl als obere als auch als untere 16 Bit.
- EAX Das Register, in das der Parameter geschrieben werden muss.

Diese Schreibweise wird im weiteren Verlauf dieser Arbeit Anwendung finden. Nach Aufruf der CPUID Instruktion sind die gewünschten Informationen in Form von 32 Bit langen Flagarrays in den Prozessorregistern EAX, EBX, ECX und EDX gespeichert und können von dort ausgelesen werden. Zur Interpretation der Flags können die Handbücher der Hersteller herangezogen werden.

```
MOV EAX, 00h
CPUID
```

Listing 1: ASM Code zur Feststellung des Manufacturer Strings

Beispiel: Eine herstellerübergreifende CPUID Funktion ist *CPUID_Fn0000_0000_EAX*. Mit dieser Funktion wird der Hersteller des Prozessors ermittelt. Nach Aufruf des Codefragments (1) befindet sich ein herstellerspzifischer, ASCII-codierter String in den Registern EBX, EDX und ECX (in dieser Reihenfolge). Im Falle eines Intel-Prozessors würde dieser String beispielsweise „GenuineIntel“ lauten [Intel]. Ein AMD-Prozessor würde „AuthenticAMD“ zurückliefern [AMD].

Der Vorteil von CPUID ist die Unmittelbarkeit der Information. Da es sich bei den gewonnen Informationen um Daten handelt, die direkt vom Prozessor stammen, ist eine Veränderung der Daten durch Zwischenschichten ausgeschlossen. Zusätzlich benötigt die

Ausführung von CPUID nur wenige Prozessorzyklen, so dass der Overhead durch den Zugriff sehr gering ausfällt.

Im Falle des Prototypen wird dieser Vorteil jedoch durch die Verwendung des *Java Native Interfaces* (siehe Kapitel 5.1.3)relativiert, das den Aufruf von CPUID kapselt und einen eigenen Overhead erzeugt (siehe Kap. 6). Eine Java-Bibliothek für den Zugriff auf Prozessorinformationen wäre SIGAR [SIGAR]. SIGAR erlaubt jedoch keine Zuordnung des aktuellen Threads zu dem logischen Prozessor, der ihn ausführt. SIGAR nutzt ebenfalls das *Java Native Interface*, so dass es auch keine Vorteile im erzeugten Overhead bietet. Dies und das Ziel des Erhalts der Plattformunabhängigkeit des Kieker-Frameworks sind die Gründe für die Verwendung von CPUID für den Prototypen.

5.1.3 Java Native Interface

Java wurde entwickelt, um plattformübergreifende Software schreiben zu können („*Write once, run anywhere*“, [JAVA97]). Um dieses Ziel zu erreichen, wird (kompilierter) Java Code (*Java Bytecode*) nicht direkt vom Betriebssystem ausgeführt. Stattdessen wird der Bytecode in einer eigenen virtuellen Maschine (*Java Virtual Machine, JVM*) ausgeführt. Die JVM bietet eine vollständige Trennung von Anwendung und Betriebssystem. So kann eine Java-Anwendung auf jeder Plattform ausgeführt werden, für die eine JVM existiert (z.B. Linux, Windows, Solaris...).

In manchen Fällen ist es jedoch notwendig, auf betriebssystem- oder plattformspezifische Funktionen oder Bibliotheken zuzugreifen. Ein Beispiel dafür wäre die Erneuerung eines bestehenden User Interfaces in Java, das seine Daten aus einer nativen Bibliothek bezieht. Die Neuentwicklung oder das Anpassen der nativen Bibliothek könnte zu schwierig oder kostenintensiv sein, um realisierbar zu sein. Für diese Fälle bietet Java das *Java Native Interface (JNI)*. JNI erlaubt es, nativen Code außerhalb der JVM auszuführen. Ein weiteres prominentes Beispiel für den Einsatz von JNI sind die Standard Java-Bibliotheken, die per JNI den Zugriff auf plattformspezifische Ressourcen wie Fensterverwaltung oder Dateisystem kapseln.

Der Prototyp nutzt diese Technik, um über in C Code eingebettete Assemblerinstruktionen Informationen über den Prozessor zu erhalten (siehe Kapitel(5.1.2). Um auf eine native Bibliothek zugreifen zu können, müssen die aus dieser Bibliothek exportierten Funktionen eine spezielle Signatur besitzen, die mit der Deklaration einer Methode mit dem Modifier *native* übereinstimmt. Dazu wird zunächst die entsprechende Methode innerhalb des Java-Codes deklariert (siehe Codeausschnitt 2).

```
public static native int sum(int a, int b);
```

Listing 2: Deklaration einer nativen Funktion in Java

Anschließend wird der Java-Code kompiliert. Mit den erstellten *.class Dateien kann nun mit einem Aufruf von *javah* automatisch eine C-Headerdatei erzeugt werden. Die erzeugte Datei enthält dann die Signatur für die native Methode (siehe Codefragment(3)).

```
JNIEXPORT jint JNICALL Java_base_Sum_sum(JNIEnv*, jclass, jint, jint);
```

Listing 3: Inhalt der erzeugten Headerdatei

Nachdem die Funktion implementiert und kompiliert wurde, kann die daraus resultierende Bibliothek in Java geladen und verwendet werden (siehe Codefragment(4)).

```
System.load("Sum");
int sumOfFourAndTwo = sum(4, 2);
```

Listing 4: Verwendung einer nativen Funktion in Java

Auf diese Weise ist auch ein Zugriff in die andere Richtung möglich, also aus nativem Code heraus in Java-Code hinein. Da dies jedoch für den Prototypen nicht relevant ist, soll diese Richtung an dieser Stelle unberücksichtigt bleiben. Eine umfassende Anleitung findet sich unter Liang [1999].

Nachteil von JNI ist, dass häufig sogenannter „Glue Code“ geschrieben werden muss. Als „Glue Code“ wird dabei der Code einer (nativen) Bibliothek bezeichnet, deren einzige Aufgabe es ist, eine Java-Schnittstelle für eine andere, bereits bestehende Bibliothek zur Verfügung zu stellen. Diese sich häufig in ähnlicher Form wiederholende Aufgabe erhöht die Komplexität und damit die Fehleranfälligkeit einer Anwendung.

Es existieren einige Alternativen zur Verwendung von „reinem“ JNI. Erwähnenswert ist hier *Java Native Access (JNA)* [JNA], das den Zugriff auf native Bibliothek komplett in Java-Code kapselt, so das kein nativer Code geschrieben werden muss. Einen ähnlichen Ansatz verfolgt *BridJ* [Chafik 2012], das durch JNA inspiriert wurde. Werkzeuge wie der *Simplified Wrapper and Interface Generator (SWIG)* [SWIG] können den „Glue Code“ automatisch erzeugen. Die hier aufgeführten Alternativen verfolgen dabei hauptsächlich den Ansatz, die Verwendung nativer Bibliotheken für den Entwickler einfacher und komfortabler zu machen. Dadurch soll die Fehleranfälligkeit der Implementierung verringert und die Entwicklungszeit verkürzt werden. Erkauft wird dies, zumindest im Falle von JNA, durch zusätzliche Funktionsaufrufe zur Laufzeit, was einen zusätzlichen Overhead bedeutet.

Für die Kieker Sonde wird reines JNI verwendet, da hier mehr Wert auf einen möglichst geringen Overhead gelegt wird. Zudem musste ohnehin eine native Methode implementiert werden, was „Glue Code“ überflüssig macht. Vorteile von SWIG u.ä. bezüglich der Komplexität und des Komforts würden kaum ins Gewicht fallen, da der zu schreibende C-Code relativ kurz und überschaubar ist.

5.1.4 AspectJ

Modularisierung und Wiederverwendbarkeit sind Grundprinzipien der objektorientierten Entwicklung. Wiederkehrende Aufgaben werden zu Funktionen und Modulen zusammengefasst. Häufig treten jedoch sogenannte *Cross Cutting Concerns* auf. Als Cross Cutting

Concerns werden wiederkehrende Aufgaben bezeichnet, die sich durch alle Schichten einer Software ziehen, bei denen aber keine der üblichen objektorientierten Ansätze zur Lösung angewendet werden kann.

Ein typisches Beispiel hierfür ist das Logging von Programmabläufen. Unabhängig davon, ob man innerhalb der Business-, Persistenz oder sonstiger Ebene einer Software operiert, ist das Loggen von Ereignissen ein wichtiger Bestandteil. Hier einen einheitlichen Logging-Mechanismus zu implementieren ist mit herkömmlichen Mitteln schwierig, da alle Schichten möglichst unabhängig voneinander sein sollen. Dies schließt Methoden wie Vererbung häufig von vornherein aus. Gelöst wird dieses Problem meist durch die Verwendung eines außenstehenden Moduls, das das Logging übernimmt. Dann ist zum einen jedes Modul der Software von diesem Modul abhängig, zum anderen muss das eigentliche Logging, also der Aufruf, das Einbinden etc., weiterhin in jedem Modul einzeln händisch erledigt werden. Zudem steigt die Komplexität des Codes, da zusätzlicher Loggingcode eingebaut werden muss, der keinen Beitrag zum Produktivcode darstellt.

Ein Ansatz, um solchen Problemen zu begegnen, ist die *aspektorientierte Programmierung (AOP)*. In der aspektorientierten Programmierung werden sogenannte *Joinpoints* verwendet, um Aufgaben wie die oben beschriebene zu übernehmen. Einen Joinpoint kann man sich hierbei als „Einhängepunkt“ vorstellen, an denen beliebiger eigener „Aspektcode“ (auch als *advice* bezeichnet) ausgeführt ausgeführt werden kann, bevor der Produktivcode weiter ausgeführt wird. Eine Menge von Joinpoints wird als *Pointcut* bezeichnet. Pointcuts werden durch bestimmte Eigenschaften definiert, beispielsweise „alle Methoden, deren Namen mit 'get' beginnen“.
Der Pointcut *"!within(ProcityProbe.*)"* umfasst alle Joinpoints, die nicht innerhalb der Klasse ProcityProbe selbst liegen. Dieses Vorgehen ist üblich, um zu vermeiden, dass der Aspektcode vor sich selbst ausgeführt wird, was zu unendlicher Rekursion und damit ggf. zum Absturz des Systems führt. Joinpoints werden unter anderem vor und nach dem Aufruf einer Methode gesetzt. Im obigen Beispiel würde zum Zwecke des Loggings beispielsweise vor der Ausführung einer Methode ihr Name sowie alle Parameter in eine Logdatei geschrieben. Im Anschluss an die Ausführung würde dann der Rückgabewert der Methode hinzugefügt. Der Produktivcode bliebe davon unberührt und der eigentliche Programmablauf ungestört. Der Vorgang, bei dem der Aspektcode in den Programmablauf eingefügt wird, wird als *Weaving* (weben) bezeichnet.

AspectJ ist eine Java-Variante der aspektorientierten Programmierung. AspectJ definiert Joinpoints mit den Bezeichnungen *Before* (vor Ausführung der Methode) und *After* (nach Ausführung der Methode). Zusätzlich ist ein *Around* Joinpoint definiert, der *Before* und *After* zusammenfasst (siehe Codefragment 5). Der Aufruf der eigentlichen Methode erfolgt dann innerhalb des Aspektcodes. In AspectJ ist dies mithilfe eines *ProceedingJoinpoint* realisiert, der Informationen über den betrachteten Joinpoint enthält. Unter den im *ProceedingJoinpoint* enthaltenen Informationen sind beispielsweise Name der aufgerufenen Methode und die Werte der ihr übergebenen Methode. Pointcuts können in AspectJ auch durch Java-Annotation beschrieben werden, z.B. „@LoggedMethod“

als Annotation zu einer Methode, die durch AspectJ Aspektcode geloggt werden soll. Aspekte werden selbst über die Annotation „@Aspect“ definiert.

```
@Aspect
public class ProcityProbe {
[...]
        @Around("!within(ProcityProbe.*)")
        public Object measure(ProceedingJoinPoint aJoinPoint) {
                // Aspectcode before execution
                [...]

                // Execution of method
                aJoinPoint.proceed();

                // Aspectcode after execution
                [...]
        }
[...]
}
```

Listing 5: Beispiel für einen AspectJ Aspekt

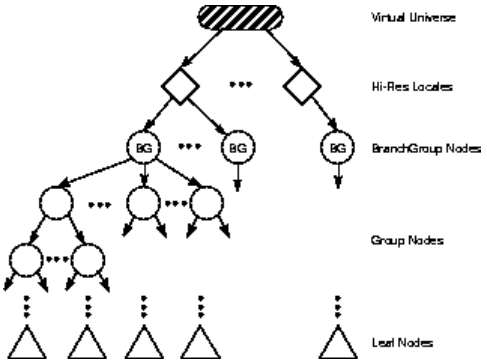
Innerhalb des Prototypen wird AspectJ verwendet, um die Kieker Sonde in eine bestehende Software weben zu können. Diese Technik bietet den gegenüber der manuellen Instrumentierung den Vorteil, dass die Software selbst nicht modifiziert werden muss. Dabei wird die Sonde genutzt um, Daten über die laufende Anwendung zu sammeln. Mehr zur Instrumentierung einer Kieker Sonde ist in Kapitel 5.4 zu finden.

5.1.5 Java3D

Für die Darstellung von 3D-Szenen, wie sie für den Prototypen benötigt werden, wird für gewöhnlich auf die *DirectX*- oder *OpenGL*-Bibliotheken zurückgegriffen, da diese inzwischen standardmäßig von allen Grafikkarten unterstützt werden. Zudem existiert eine große Entwicklergemeinde und eine umfangreiche Dokumentation für diese Bibliotheken. Java3D ist eine Kapselung von DirectX und OpenGL für Java. Ziel von Java3D ist es dabei nicht, das Interface der entsprechenden Bibliotheken 1:1 abzubilden, sondern ein einfacheres, aber ausreichendes Interface zu bieten. Java3D bietet Funktionen, die das bequeme Erstellen einfacher Objekte (Würfel, Kugeln...) ermöglicht. Unter anderem bietet Java3D Funktionen an, die das Positionieren der Kamera erleichtern oder dem Entwickler für Standardfälle sogar abnimmt. Zudem können mit wenig Aufwand Animationen und Interaktionen erstellt werden.

Umfassender Container einer Szene ist das Java3D-*Universum*, das durch ein dreidimensionales Koordinatensystem gegeben ist. Jede hochauflösende Koordinate besteht aus drei 256 Bit langen Fixpunktzahlen. Dies ermöglicht eine maximale Größe des Universums von 2^{87} Metern, umgerechnet ca. 20 Milliarden Lichtjahre, bei einer Auflösung von 2^{-115} Metern [j3d 1999].

Abbildung 11: Java3D Virtual Universe. Quelle: [j3d 1999]



Eine Szene in Java3D wird intern als Baum modelliert (siehe Abb. 11). Das Universum ist hierbei die Wurzel des Baumes, der ein oder mehrere *Locale*-Knoten angehängt sind. Die Position der Locale-Knoten werden über die hochauflösenden Koordinaten angegeben. Ein Locale-Knoten dient als „Anker“ für weitere Objekte, deren Koordinaten relativ zu denen des Locale-Knotens sind. Er dient sozusagen als Ursprung für ein eigenes Koordinatensystem. Dadurch ist es möglich, die Koordinaten aller weiteren Objekte als Fließkommazahlen mit geringerer Auflösung anzugeben.

Unterhalb eines Locale-Knotens sind *BranchGroup*-Knoten eingehängt. BranchGroup-Knoten dienen als „Compile Unit“ [j3d 1999], d.h. ein Container dessen Inhalt eine Menge darstellbarer Objekte ist.

Es existieren verschiedene Typen von *Group*-Knoten, die an einen BranchGraph-Knoten angehängt werden können. Der für den Prototypen wichtigste Knoten ist der *TransformGroup*-Knoten. Er erlaubt die komfortable Transformation (d.h. Translation, Rotation, Skalierung...) seiner Kinder.

Leaf-Knoten sind eine Zusammenfassung von geometrischen Objekten, Lichtern und Tönen, die in der Szene vorkommen.

Das Codebeispiel 6 zeigt ein „Hello World“ Programm für Java3d. Der abgebildete Code erzeugt ein Universum *universe* und eine BranchGroup *group*. Anschließen wird das tatsächlich sichtbare Objekt zu *group* hinzugefügt, ein generischer *ColorCube*. ColorCube ist eines der von Java3D zur Verfügung gestellten Objekte, das einen Würfel mit voneinander verschiedenen Seitenfarben darstellt. Bevor *group* dem Universum hinzugefügt wird, wird die Kameraposition direkt auf den Würfel gesetzt. Über die *main()* Methode kann das Programm gestartet werden. Abbildung 12 zeigt die Ausgabe des Programms.

```
[...]
public class Hello3d {

public Hello3d() {
```

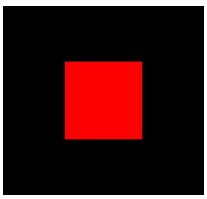


Abbildung 12: Ausgabe der Hello3d Klasse, Quelle:[Hopkins 2001]

```
    SimpleUniverse universe = new SimpleUniverse();
    BranchGroup group = new BranchGroup();
    group.addChild(new ColorCube(0.3));
    universe.getViewingPlatform().setNominalViewingTransform();
    universe.addBranchGraph(group);
}

public static void main( String[] args ) {
   new Hello3d();
}

} // end of class Hello3d
```

Listing 6: Hello3d Klasse, Quelle:[Hopkins 2001]

Es existieren verschiedene Alternativen zu Java3D, darunter kostenlose Varianten wie JOGL[JOGL], Xith3D[Yazel et al. 2012] und Aviatrix3d[Couch et al. 2012], sowie einige kostenpflichtige Bibliotheken.

Java3D wurde als Werkzeug für den Prototypen gewählt, da es als Produkt direkt von Sun bzw. Oracle entwickelt wurde. Dadurch existiert ein reichhaltiges Angebot an Tutorials und API-Dokumentation sowie eine große Entwicklergemeinde. Weiter sind die oben genannten Funktionen zum einfachen Erstellen von Objekten und die Abstraktion als Szenengraph ein deutlicher Vorteil gegenüber 1:1 Umsetzungen der OpenGL API, wie z.B. JOGL. Beispielsweise nutzt der Prototyp zur Darstellung der Säulen die Java3D Klasse *Box*, und die Animationen sind durch Java3D *Behaviors* realisiert.

5.2 Visuelles Design

In diesem Kapitel soll die visuelle Gestaltung des Prototypen beschrieben werden. Zwar sollte ein gutes Werkzeug stets nach dem Motto „Form folgt Funktion“ entworfen werden, jedoch kann ein durchdachtes optisches Design einen großen Beitrag zum Nutzen des Werkzeugs leisten. Dies gilt besonders für ein Werkzeug, das abstrakte Daten wie Zeiten und Prozessorauslastungen in eine Form bringen soll, die für ein von den Augen gesteuertes Tier wie den Menschen übersichtlich und leicht verständlich ist.

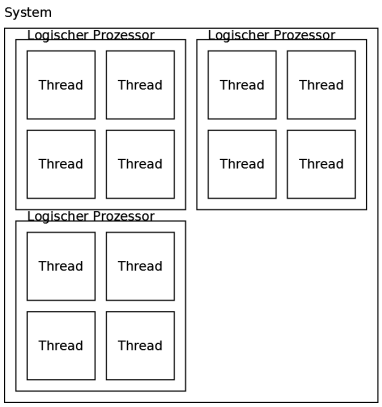


Abbildung 13: Layout der Visualisierung

5.2.1 Layout

Das Layout der Visualisierung ist bewusst simpel gehalten, da mit wachsender Anzahl angezeigter Threads und logischer Prozessoren schnell eine große Menge von Objekten überblickt werden muss. Die oberste Ebene des Layouts ist die *Stadt*, also die Gesamtheit aller angezeigten *Blöcke*. Eine Stadt repräsentiert ein komplettes Multiprozessorsystem, in dem jeder logische Prozessor durch einen Block dargestellt wird. Die Begriffe sind an eine City Metapher (siehe Kap. 3.6) angelehnt. Jeder Block besteht aus einem Label, das die Id(s) des zugehörigen logischen Prozessors angibt, sowie aus einer Menge von *Häusern*, von denen jedes einen Thread darstellt (siehe Abbildung 13). Ein Haus stellt zum einen durch seine Höhe und zum anderen durch eine textuelle Anzeige auf dem Dach die aktuelle Laufzeit des zugehörigen Threads dar (siehe Kapitel 5.6.1). Zusätzlich zur Laufzeit werden auf dem Dach des Hauses auch der Name des dargestellten Threads sowie die *Hungriest Method* angezeigt. Die *Hungriest Method* ist die zu diesem Zeitpunkt am Häufigsten aufgerufene Methode. Der Prototyp bietet zur Zeit nicht die Möglichkeit, die *am längsten Laufende* Methode zu ermitteln und darzustellen (siehe auch Kapitel 7.3).

Innerhalb der Stadt sind alle Blöcke anhand eines quadratischen Rasters ausgerichtet. Die Position der Blöcke wird bestimmt, indem zunächst die Anzahl der Blöcke B bestimmt wird. Anschließend wird die Seitenlänge des Rasters (in Feldern) bestimmt. Dies geschieht, indem $\sqrt{B}$ berechnet und ggf. aufgerundet wird. Dadurch wird sichergestellt, das ausreichend Felder für alle Blöcke zur Verfügung stehen. Anschließend werden die Blöcke der Reihe nach auf die Felder verteilt. Die Aufteilung der Häuser innerhalb eines Blocks geschieht analog dazu. Als zusätzliche Orientierung wird ein Gitter am Boden der Darstellung angezeigt.

Zur Zeit werden pro Block jederzeit alle Threads angezeigt, die während der Messung gefunden wurden. Im Vorfeld und während der Implementierung des Prototypen wurde

eine Alternative zu dieser Darstellung erwogen. Dabei würden innerhalb eines Blocks nur die Häuser angezeigt, die einen zur Zeit laufenden Thread repräsentieren. Vorteil dieser Darstellung ist, dass weniger Objekte gleichzeitig angezeigt werden. Ein Nachteil ist jedoch, dass ein Thread keinen festen Platz innerhalb eines Blocks hat. Dadurch muss der Anwender nach jedem Threadwechsel erneut den beobachteten Thread innerhalb des Blocks suchen. Eine Lösung dafür wäre das Animieren von Threadwechseln, beispielsweise durch das verschieben der wechselnden Threads von einem Block zum nächsten. Auch dies kann jedoch unübersichtlich werden, wenn viele Threads häufig wechseln. Der Prototyp kann die Verwendung von Threadpools nicht direkt erkennen. Er beobachtet jeden Thread über die gesamte Lebenszeit der Anwendung und wird entsprechend alle Laufzeiten und Methodenaufrufe aufaddieren, auch wenn der entsprechende Thread inzwischen eine andere Aufgabe hat. Dies kann zum Beispiel bei Serveranwendungen vorkommen, wo ein Thread für mehrere Verbindungen „recycelt“ wird.

Die aktuelle Implementierung hat den Vorteil weniger beweglicher Teile, was die Orientierung innerhalb der Visualisierung verbessert. Dies gilt jedoch nur bis zu einer bestimmten Anzahl von Threads. Wird diese Anzahl überschritten, sind zu jeder Zeit zu viele Objekte sichtbar, um noch überschaubar zu sein. Viele Anwendungen nutzen sogenannte „Threadpools“, d.h. sie starten im Laufe ihres Lebens nur eine begrenzte Anzahl an Threads, anstatt stets neue zu erzeugen. Dies kann indirekt auch die Übersichtlichkeit der Visualisierung verbessern.

5.2.2 Farben

Der Prototyp nutzt einfache, quaderförmige Säulen zur Darstellung von Threads. Zur Unterscheidung einzelner Threads wird der Threadname angezeigt, wie er von der Kieker Sonde ermittelt wird. Um die Zeit, die benötigt wird, um einen Thread zu finden, und somit die Übersichtlichkeit zu erhöhen, sind die Säulen individuell eingefärbt. Die Farbe einer Säule ergibt sich aus dem Namen des Threads, den die Säule repräsentiert. Dazu wird über den Threadnamen eine 32 Bit lange MD5-Prüfsumme gebildet. Danach werden die Bytes 0, 1 und 2 der Prüfsumme jeweils durch $(2^8) - 1$, also 255, geteilt, um eine Fließkommazahl zwischen 0 und 1 zu erhalten. Die so ermittelten Werte werden dann den Farben Rot, Grün und Blau zugewiesen (in dieser Reihenfolge). Das vierte Byte wird verworfen. Abbildung 7 zeigt den kompletten Algorithmus in Java.

```
private static Color3f colorForName(String aName) {

    // Get the MD5 hash of the string.
    byte[] byteHash = { 0, 0, 0 };

    MessageDigest digest = MessageDigest.getInstance("MD5");
    byteHash = digest.digest(aName.getBytes());

    // Get the raw values from the byte array.
```

```
        float redRaw = new Byte(byteHash[0]).floatValue();
        float greenRaw = new Byte(byteHash[1]).floatValue();
        float blueRaw = new Byte(byteHash[2]).floatValue();

        // Divide by 255 to bring values in a range from 0 to 1.
        float red = Math.abs(redRaw / 255.0f);
        float green = Math.abs(greenRaw / 255.0f);
        float blue = Math.abs(blueRaw / 255.0f);

        // Create and return Color
        Color3f returnColor = new Color3f(red, green, blue);
        return returnColor;
    }
```

Listing 7: Farbgewinnung durch MD5-Prüfsumme

Da sich Threadnamen nicht oder nur selten ändern, bleibt die Farbe eines Threads für jeden Durchgang gleich, da sich die MD5-Prüfsumme nicht ändert, solange der Threadname gleich bleibt. Dies ist ein Vorteil, wenn beispielsweise ein bestimmter Thread Ziel einer Optimierung ist, da er leichter wiederzuerkennen ist („wir müssen den hellgelben Thread im Auge behalten“).

Die Stärken dieser Vorgehensweise treten besonders bei kleineren Threadzahlen, bis ca. 100 Threads, hervor. Darüber kann es zu Überschneidungen in der Farbe kommen. Zwar wird ein 24 Bit großer RGB-Farbraum genutzt, die Anzahl der für Menschen unterscheidbaren Farben ist jedoch geringer. Eine theoretische Anzahl an für Menschen unterscheidbaren Farben liegt bei 150 Farben in ca. 600 verschiedenen Helligkeiten (Völz [1999], S.42). Da dies von Anwender zu Anwender variiert und zusätzliche Einschränkungen durch das Anzeigegerät entstehen (z.B. schlechte Kontrastwerte oder falsch eingestellte Helligkeit) ist eine Schätzung der oben genannten Grenze von ca. 100 Threads mit unterschiedlichen Farben sinnvoll.

5.3 Architektur

Der Prototyp ist in Java implementiert, um eine Anbindung an das Kieker Framework zu erleichtern. Er ist in drei aufeinander aufbauende Teile geteilt (siehe Abb. 14). Im ersten Teil, der Datenerfassung, werden von der Kieker Sonde Daten zur späteren Auswertung erfasst. Abbildung 27 zeigt das Klassendiagramm der Datenerfassung. Im zweiten Teil, dem *Playback*- oder Datenauswertungsteil, werden die von der Sonde gelieferten Daten aufbereitet und dem dritten Teil zur Verfügung gestellt. Das Klassendiagramm der Datenauswertung befindet sich in Abbildung 28. Der dritte Teil besteht aus der eigentlichen Visualisierung, in dem die erfassten und aufbereiteten Daten als dreidimensionales Balkendiagramm angezeigt werden. Die Abbildungen 29 und 30 (siehe Anhang, Kapitel 9.4) zeigen die Klassen dieses Parts. Die einzelnen Komponenten werden in den nachfolgenden Kapiteln (Kap. 5.4, 5.5, 5.6) genauer beschrieben.

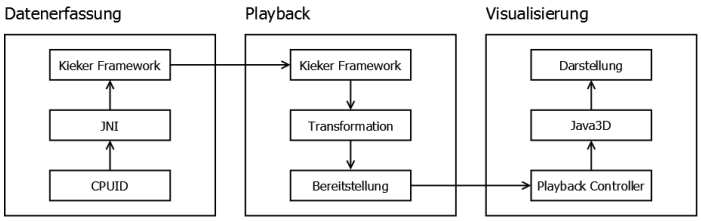


Abbildung 14: Architektur

Diese Einteilung erlaubt es, einzelne Bestandteile des Prototypen weitgehend unabhängig voneinander zu gestalten und, falls notwendig, auszutauschen. Zudem schafft es eine klare Trennung der Aufgaben und Funktionen der einzelnen Komponenten.

Die Trennung von Datenerfassung und Playback ergibt sich durch die Ablage der Daten im Dateisystem. Da ihre Ausführung zeitlich voneinander unabhängig ist, stellt dies die einzige Möglichkeit dar, die gesammelten Daten auszutauschen. Überdies müssen das System, auf dem die Daten gesammelt werden (auf dem eine Anwendung mit implementierter Sonde läuft) und das System, auf dem die Daten ausgewertet werden (auf denen eine Instanz der Visualisierung läuft), nicht zwangsläufig identisch sein. Tatsächlich werden im praktischen Einsatz eher zwei unterschiedliche Systeme verwendet (z.B. ein Server zur Datenerfassung und ein Desktoprechner zur Analyse).

Playback und Visualisierung bilden zur Laufzeit des Prototypen eine Einheit, sind jedoch architektonisch durch Interfaces voneinander abstrahiert. Dadurch kann das GUI einfach gehalten werden, d.h. das GUI muss nichts über die Bedeutung der Daten wissen, die sie darstellt. Als Verbindungsstück dient der *PlaybackController*, der im Visualisierungspart untergebracht ist. Er dient, zusammen mit der Klasse *Main* als zentraler Knoten, der die eigentliche Visualisierung mit dem Playback und einigen zusätzlichen GUI-Elementen verbindet. Die zusätzlichen GUI-Elemente dienen zur Bedienung des Prototypen.

Klassen, auf die sämtliche Parts zugriff haben müssen, sind in einem *commons* Paket abgelegt (siehe auch Abbilfung 31. Hier befindet sich die Klasse *CPUInfo*, eine reine Datenhaltungsklasse, die alle drei Ids (Package, Core, SMT, siehe Kapitel 5.4.1) eines logischen Prozessors hält.

5.4 Die Kieker Sonde

Zur Analyse und Darstellung von Prozessorauslastungen werden zunächst Daten über die Laufzeiten einzelner Methoden bzw. Threads benötigt. Kernstück der Datenerfassung des Prototypen ist die hierfür erstellte Sonde in Kombination mit dem Kieker Framework. Das folgende Kapitel beschreibt die Sonde und die von ihr gelieferten Daten.

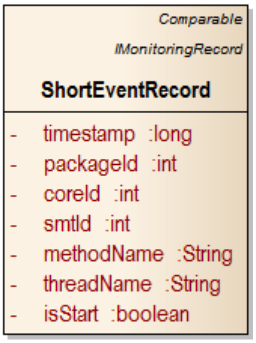


Abbildung 15: ShortEventRecord

5.4.1 Aufbau und gelieferte Daten

Die ProcityProbe erbt von der bereits im Kieker Framework enthaltenen abstrakten Klasse AbstractAspectJProbe. Ihre Funktion ist über einen *Around* Einhängepunkt realisiert, der ein *ProceedingJoinPoint* übergeben wird. Der *Around* Einhängepunkt ermöglicht es, sowohl den *Before* als auch den *After* Einhängepunkt innerhalb einer einzelnen Methode abzuarbeiten (siehe auch Kapitel 5.1.4). Dies erleichtert die Verwaltung und Speicherung von Daten, da Werte innerhalb der Methode aufbewahrt werden können.

Die eigentliche Funktion der Sonde ist das ermitteln von Daten, die jeweils vor und nach Ausführung einer gemessenen Methode in ein *ShortEventRecord* geschrieben werden. Anschließend wird der Record dem Kieker MonitoringController zur weiteren Bearbeitung übergeben. Der *ShortEventRecord* (siehe Abbildungg 15) ist ein Kieker Record; er implementiert das Interface *IMonitoringRecord*. Dadurch kann sie vom Kieker Framework verwaltet werden. Er wird von der Sonde mit folgenden Daten gefüllt:

Zeitstempel Der Zeitpunkt, zu dem die Messung erfolgt ist, also jeweils Start- oder Endzeit der gemessenen Methode. Die Zeiten werden durch die vom Kieker Framework zur Verfügung gestellte *ITimeSource* ermittelt. Die Sonde speichert den Wert ohne die Angabe einer Zeiteinheit, da *ITimeSource*s im Kieker Framework austauschbar sind.

Prozessor Id Die Id des zum Messzeitpunkt ausführenden Prozessors. Die Id besteht aus drei Teilen, der *SMT Id (Simultaneous Multithreading Id)*, die die ID des logischen Prozessors angibt, der *CoreId*, die den Prozessorkern angibt, dem der logische Prozessor zugeordnet ist, sowie der *Package Id*, die den physikalischen Prozessor in einem Multiprozessorsystem identifiziert. Dabei ist zu beachten, das die SMT Id nur innerhalb eines Cores und die Core Id nur innerhalb eines Packages eindeutig ist. Durch Angabe aller drei Teile kann der ausführende logische Prozessor eindeutig identifiziert werden. Alle drei Ids werden durch den Aufruf einer nativen Methode durch ein JNI (siehe Kapitel 5.1.3) ermittelt. Innerhalb der nativen Methode wird CPUID (siehe Kapitel 5.1.2) benutzt, um

die Ids direkt vom Prozessor zu erfragen. Das im Prototypen benutzte Verfahren, um die Ids aus einem Intel-Prozessor zu extrahieren, wird in Kapitel 5.4.4 genauer beschrieben.
Methodenname Der voll qualifizierte Name der gemessenen Methode inklusive Paket- und Klassenname. Die Namen können über den durch AspectJ zur Verfügung gestellten *ProceedingJoinPoint* ermittelt werden.
Threadname Der Name des Threads, in dem die gemessene Methode ausgeführt wird. Der Threadname kann über die Java-eigene Methode *Thread.getName()* erfragt werden.
Methodenzustand Dieser boolesche Wert ist ein Hilfsparameter, der angibt, ob die Messung am Anfang oder am Ende der Methode geschehen ist. Der Methodenzustand wird in Form einer Variable gespeichert, anstatt zwei sehr ähnliche Klassen zu erzeugen. Beide Klassen würden die gleichen Daten enthalten, lediglich der Klassenname würde sich unterscheiden. Die Pflege zweier sehr ähnlicher Klassen würde den Verwaltungsaufwand innerhalb der Software erhöhen. Der Methodenzustand wird später bei der Erstellung der Playbackdaten (siehe Kap. 5.5) ausgewertet.

5.4.2 Instrumentierung

Um Daten über ein Softwaresystem liefern zu können, muss die Sonde zunächst in das System eingebettet werden. Zur Instrumentierung der Sonde ist AspectJ (siehe Kap. 5.1.4) vorgesehen. Dazu müssen keine Änderungen am Sourcecode des untersuchten Systems vorgenommen werden. Die Sonde kann über die Parametrierung der Java VM während des Programmstarts eingebunden werden. Codelisting 8 ist ein Beispiel für die Instrumentierung in eine Software mittels einer XML-Datei. Es zeigt, wie der Aspekt *kieker.monitoring.probe.aspectJ.ProcityProbe*, also die Sonde des Prototypen in die Klasse *bookstoreTracing.BookstoreStarter* eingebettet wird. Auf diese Weise kann der Aspektcode zur Laufzeit in den Produktivcode eingewoben werden.

```
<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN"
        "http://www.aspectj.org/dtd/aspectj_1_5_0.dtd">
<aspectj>
    <weaver options="">
        <include within="bookstoreTracing.BookstoreStarter"/>
    </weaver>

    <aspects>
      <aspect name="kieker.monitoring.probe.aspectJ.ProcityProbe"/>
    </aspects>
</aspectj>
```

Listing 8: Beispiel aop.xml

Details zu AspectJ findet sich unter 5.1.4.

5.4.3 Meßungenauigkeit bei Prozessorwechsel

Kapitel 5.1.1 und 5.1.4 beschreiben die der Instrumentierung der Sonde zugrundeliegende Technik. Ein Merkmal dieser Art der Instrumentierung ist, das die Einhängepunkte der Sonde nur an zwei Punkten liegen, vor und hinter der Ausführung einer Methode. Während dies für die Ermittlung der Laufzeit ausreichend ist, ergibt sich für die Zuordnung des Threads, in dem die gemessene Methode aufgerufen wurde, zu dem logischen Prozessor, der sie ausführt, ein Problem: Der ausführende logische Prozessor kann nur an zwei Stellen bestimmt werden, und die Prozessoren am Anfang und zum Ende müssen nicht identisch sein. Da das Scheduling eines Threads für den Thread selbst transparent geschieht, ist es mit dieser Methode nicht möglich, Aussagen über den Verbleib des Threads zwischen diesen beiden Messpunkten zu treffen. Es ist also möglich, das ein Thread zwischen zwei Messzeitpunkten beliebig oft und beliebig lange auf anderen logischen Prozessoren ausgeführt wurde. Daraus ergibt sich eine Ungenauigkeit bei der Erhebung der Daten, insbesondere bei der Zuordnung der Threadlaufzeiten zu den einzelnen logischen Prozessoren. Für den weiteren Aufbau des Prototypen bedeutet das, das ein sinnvoller Weg gefunden werden muss, mit dieser Ungenauigkeit umzugehen. Die verwendete Lösung wird in Kapitel 5.5.1 beschrieben.

5.4.4 Unterstützte Prozessoren

Die Assembler Instruktion CPUID bietet einige herstellerübergreifende Funktionen, darunter *CPUID_Fn0000_0001_EAX*, die eine 8-Bit Id des logischen Prozessors bereitstellt. Diese Funktion ist jedoch in ihrer Aussagekraft beschränkt, da sie keine Rückschlüsse auf die Id des momentanen *physikalischen* Prozessors zulässt. Bei der Ermittlung der Prozessortopologie, also auch der Ermittlung des momentanen physikalischen Prozessors, unterscheiden sich jedoch die von den Herstellern vorgesehenen Methoden.

Beispielsweise stellt Intel mit der Funktion *CPUID_Fn0000_000B_EAX* auf einigen Prozessoren eine Funktion zur Verfügung, die eine 32- statt 8-Bit Id zur Verfügung stellt, in der auch die Id des physikalischen Prozessors enthalten ist [Intel]. Mithilfe dieser Funktion kann eine Karte eines Multiprozessorsystems aufgebaut werden (siehe auch Kuo [2009]).

AMD geht hier einen anderen Weg und verpackt auch die Id des physikalischen Prozessors in die von *CPUID_Fn0000_0001_EAX* gelieferte Id. Um an die Id des physikalischen Prozessors zu gelangen, muss jedoch erst eine Bitmaske generiert werden, für die Daten aus der Funktion *CPUID_Fn8000_0008_EAX* benötigt werden.

Da der Zugriff auf die Id des physikalischen Prozessors nicht einheitlich ist, unterstützt die für den Prototypen verwendete Sonde nur Intel-Prozessoren. Eine Anpassung oder Erweiterung für andere Prozessoren ist jedoch möglich (siehe auch Kapitel 7.1).

Aus der 32-Bit Id, die von der Funktion *CPUID_Fn0000_000B_EAX* zurückgeliefert wird, lassen sich insgesamt drei Ids für drei unterschiedliche Ebenen in der Prozessor-

hierarchie auslesen. Dazu wird zusätzlich ein Parameter im ECX Register gesetzt, der die gewünschte Ebene angibt. Zur Auswahl stehen dabei die Thread bzw. SMT Ebene (ECX=0), Core Ebene (ECX=1) und die Package Ebene (ECX=2). Nach der Ausführung von CPUID enthält das EAX Register, in den Bits 0 bis 4, einen Shift-Wert. Der Shift-Wert gibt die Anzahl Bits an, um die die in EDX enthaltene (Gesamt-)Id nach rechts geschoben werden muss, um die zu der gewählten Ebene gehörige Id zu erhalten. Die Shift-Werte müssen nur einmal zum Beginn der Messreihe ermittelt werden und können dann für den späteren Gebrauch zwischengespeichert werden. Während der Messung wird nur *CPUID_Fn0000_000B_EAX* und die entsprechenden Shift-Werte angewendet. Wird *CPUID_Fn0000_000B_EAX* vom Prozessor nicht unterstützt, so wird eine UnsupportedCpuException geworfen, die eine entsprechende Fehlermeldung enthält. Der Prototyp führt diese Logik auf Java-Ebene aus, um die Funktionalität des durch JNI (siehe Kapitel 5.1.3) angesprochenen nativen Parts der Implementierung möglichst klein zu halten.

5.5 Der Playback-Part

Der Prototyp stellt Messdaten innerhalb von frei wählbaren Intervallen dar. Im Gegensatz dazu sind die Messpunkte, die von der Sonde geliefert werden, unsortiert und unregelmäßig. Die von der Sonde ermittelten Daten müssen also zunächst aufbereitet werden. Dies geschieht im *Playback*-Part des Prototypen.

Im Playback-Part werden von einer Sonde erstellte Kieker MonitoringRecords aus einen Dateisystem eingelesen und in eine Form gebracht, die eine Interpolation von fehlenden Messpunkten ermöglicht. Auf diese Weise kann dem Visualisierungspart des Prototypen später ein lückenloser Datensatz angeboten werden. Der Playback-Part ist unterteilt in zwei Parts: Transformation und Bereitstellung der Daten.

5.5.1 Transformation

Dieser Teil, dessen Implementierung im Paket *playbackTransform* zu finden ist, wird nur einmal während des Einlesens der Sondendaten ausgeführt. Hier werden zuerst die aufgezeichneten *ShortEventRecords* (siehe Kapitel 5.4) eingelesen, nach Threadnamen getrennt und nach ihrem Zeitstempel sortiert (siehe Abbildung 16).

Anschließend werden die Records für die einzelnen Threads zu einer zeitlich sortierten Liste von *ThreadStateRecord*s (im weiteren Verlauf auch *Timeline*) akkumuliert (siehe Abbildung 18). *ThreadStateRecord*s (siehe Abbildung 17) enthalten Daten, die den momentanen Zustand des Threads repräsentieren.

Die wichtigsten davon sind:

Zeitstempel Der Zeitpunkt, den dieser Datensatz repräsentiert.

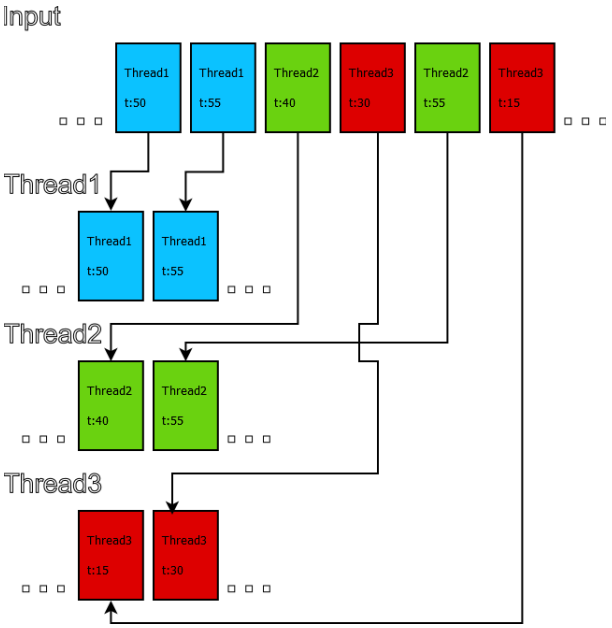


Abbildung 16: Erster Transformationsschritt

Gesamtlaufzeiten des Threads pro logischem Prozessor Eine Liste, in der für jeden logischen Prozessor die Gesamtlaufzeit des Threads gespeichert ist. Die Gesamtlaufzeit ist Summe der einzelnen gemessenen Laufzeiten seit Beginn der Messung.
Aufzeichnung über die in diesem Thread ausgeführten Methoden Eine Liste mit Methodenzuständen, in denen hauptsächlich die Gesamtzahl der Aufrufe dieser Methode (in diesem Thread) seit Beginn der Messung gezählt wird.

Die Timeline wird durch schrittweise Verarbeitung der im ersten Schritt erstellten Listen von ShortEventRecords erstellt. Ein Schritt wird dabei durch einen Record in der Liste angegeben. Als erstes wird ein leerer *ThreadStateRecord* R_{base} angelegt (vgl. Abbildung 18). In jedem darauf folgenden Schritt wird für R_{base} :

1. die nächste ShortEventRecord ausgewählt
2. die Differenz t_{delta} zwischen dem Zeitstempel des aktuellen ShortEventRecord, $t_{current}$, und den Zeitstempel des zuletzt gesehenen ShortEventRecord, t_{last}, auf die Gesamtlaufzeit des zuletzt gesehenen logischen Prozessors $t_{cpu_{last}}$ addiert
3. der zuletzt gesehene logische Prozessor cpu_{last} aktualisiert

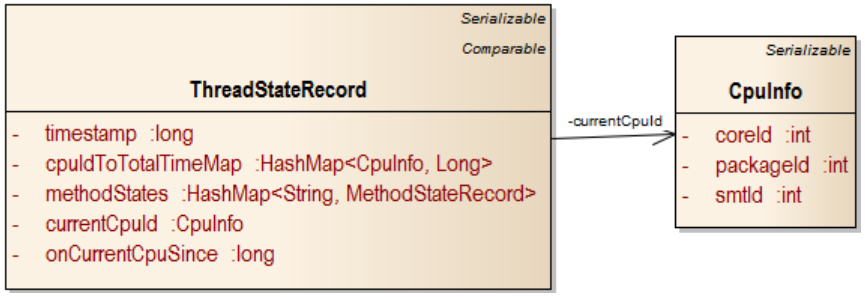


Abbildung 17: ThreadStateRecord

4. die Gesamtaufrufe für die gemessene Methode inkrementiert, falls es sich um den Anfang der Messung handelt (nicht im Bild).

Nachdem alle Werte des aktuellen *ShortEventRecord* verarbeitet wurden, wird eine Kopie des aktualisierten R_{base} der Timeline angehängt. Auf diese Weise entsteht eine Liste von Datensätzen, von denen jeder einzelne die akkumulierten Daten aller *ShortEventRecord*s seit dem Anfang der Messung enthält.

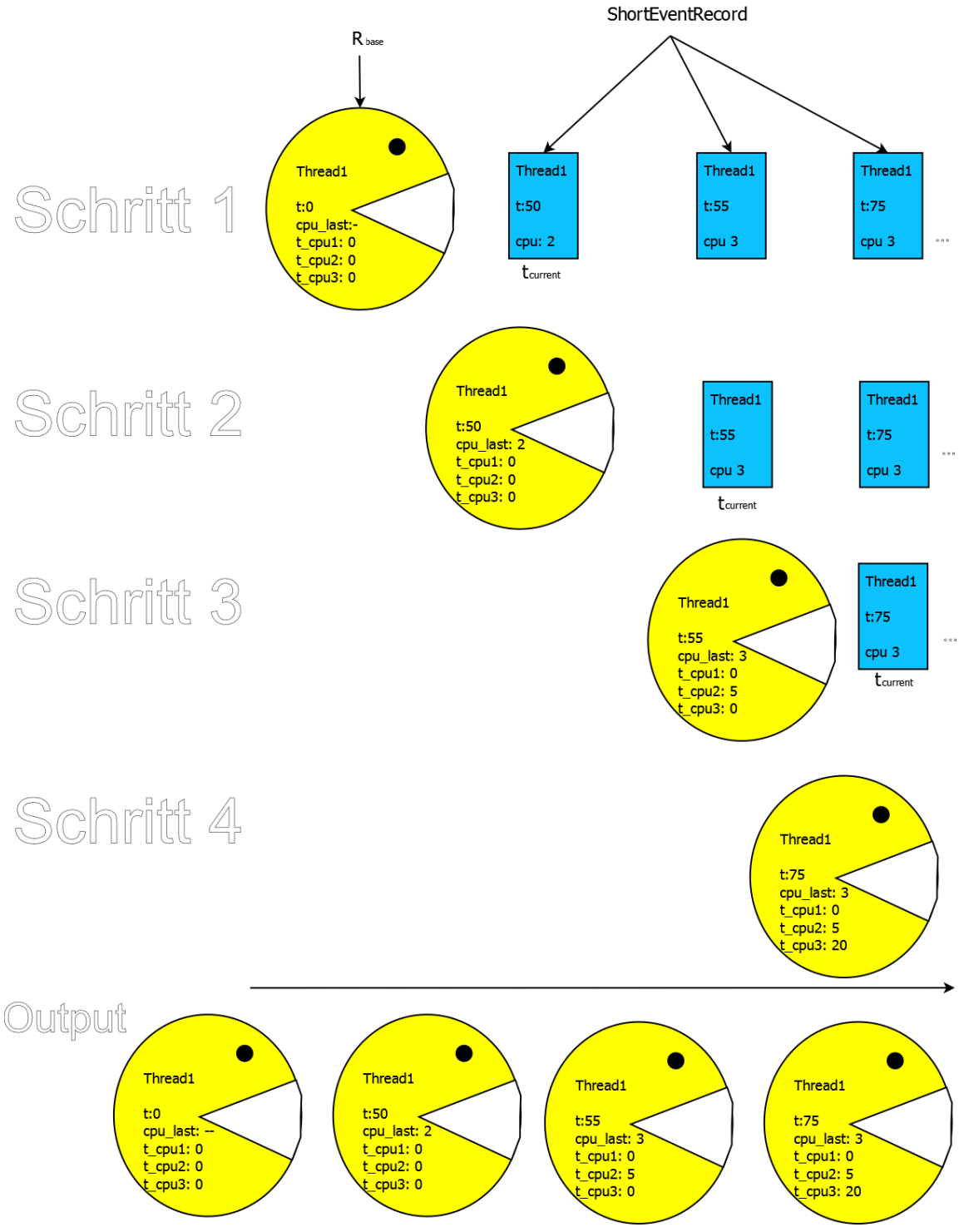


Abbildung 18: Zweiter Transformationsschritt (pro Thread)

Aus der beschriebenen Transformation lässt sich bereits die gewählte Lösung für das in Kapitel 5.4.3 beschriebene Problem ableiten. Es wird immer davon ausgegangen, dass ein Thread stets die volle Zeit bis zur Entdeckung eines anderen logischen Prozessors genutzt hat. Wird also beispielsweise zum Zeitpunkt 50 der logische Prozessor mit der Id 1 aufgefunden und zum Zeitpunkt 75 der Prozessor mit der Id 2, so wird angenommen, dass der Thread 25 Zeiteinheiten lang auf dem Prozessor mit der Id 1 lief. Genauso wäre es denkbar, die Laufzeit dem neu gefundenen logischen Prozessor zuzuschlagen oder die Laufzeit nach einem festen Schema aufzuteilen (z.B. halbe-halbe). Da jedoch keine generellen Regeln für den Threadwechsel zwischen den Messzeitpunkten ableitbar sind, sind alle drei Möglichkeiten gleichwertig, bieten also weder Vor- noch Nachteile. Daher wurde die verwendete Lösung willkürlich im Laufe der Implementierung gewählt.

Das Problem wird weiter gemindert durch die Auswertung des momentanen logischen Prozessors am Anfung *und* am Ende der gemessenen Methode. Da innerhalb von Methoden oft weitere Methoden aufgerufen werden, ergeben sich „verschachtelte“ Messungen, die zu einer höheren Genauigkeit führen. Abbildung 19 veranschaulicht dieses Prinzip. Es zeigt den Aufruf einer Methode B. Methode B ruft im Laufe seiner Ausführung Methode C auf. Jeweils an den Aufruf- und Rückkehrpunkten von B und C ist der von der Sonde ermittelte logische Prozessor aufgetragen. Zusätzlich ist die Zeit zwischen zwei Messpunkten in generischen Zeiteinheiten (ZE) aufgetragen. Im abgebildeten Verlauf befindet sich der Thread zu Beginn der Ausführung von B auf dem logischen Prozessor CPU1 (Messpunkt m1). Beim Aufruf von C (Messpunkt m2) jedoch wird CPU2 als logischer Prozessor ermittelt. Während der Ausführung wurde also der Thread von CPU1 auf CPU2 verschoben. Nach der oben erläuterten Konvention werden die 10ZE, die zwischen den beiden Messpunkten ermittelt wurden, der Gesamtlaufzeit auf CPU1 zugerechnet. An Messpunkt m3 wurde der Thread wieder auf CPU1 verschoben. Dennoch wird die ermittelte Laufzeit von 15ZE auf die Gesamtlaufzeit von CPU2 addiert. Die 10ZE zwischen m3 und m4 werden wieder CPU1 zugeschlagen. Daraus ergibt sich für diesen Aufruf eine Laufzeit von 20ZE auf CPU1 und 15ZE auf CPU2.

Würde der momentane logische Prozessor nur einmalig für eine Methode aufgerufen, beispielsweise am Ende ihrer Ausführung, so wäre der abgebildete Wechsel an Messpunkt m2 übersehen worden. Als Resultat wäre eine Laufzeit von 35ZE auf CPU1 und 0ZE auf CPU2 ermittelt worden.

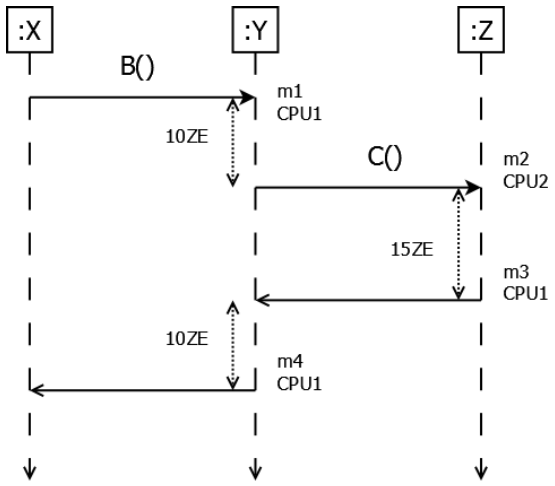


Abbildung 19: Erhöhte Genauigkeit durch verschachtelte Messung

5.5.2 Bereitstellung

Nachdem die gesammelten Daten wie oben beschrieben verarbeitet wurden, können sie der nächsten Ebene zur Verfügung gestellt werden. Dies geschieht über eine Implementierung des *IPlaybackProvider* Interfaces. Die Visualisierung (oder ein anderer Abnehmer) kann über diese Schnittstelle beliebige Intervalle aus der Timeline des IPlaybackProviders anfordern. Dazu verfügt der *PlaybackController* über eine Liste von *IPlaybackController*n, einem pro Thread. Als Rückgabe erhält sie eine *ThreadStateRecord*. Diese *ThreadStateRecord* beschreibt den Zustand des Threads, als ob der Anfang der Messung dem Beginn des angeforderten Intervalls entspräche.

Um dieses Verhalten, insbesondere die Wahl von Start- und Endpunkten, zu denen keine Messwerte vorliegen, zu realisieren, müssen zunächst zwei Interpolationen durchgeführt werden. Es werden zwei *ThreadStateRecord*s benötigt, eine für den Anfang des angeforderten Intervalls, eine für das Ende. Die Interpolation beider Datensätze läuft gleich ab:

$t_{current}$ = gewünschter Zeitstempel
if Datensatz r_{exist} für Zeitpunkt $t_{current}$ existiert **then**
  return r_{exist}
else
  $r_{interpolated}$ = neuer Datensatz
  r_{base} = nächstkleinerer Datensatz

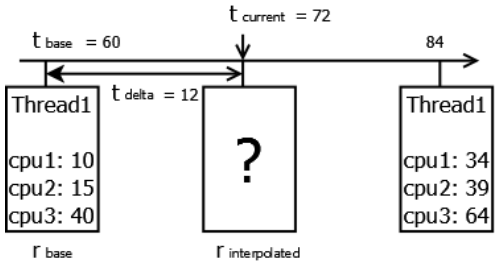


Abbildung 20: Interpolationsbeispiel

	r_{base}	t_{delta}	$r_{interpolated}$
t_{cpu_1}	10	12	22
t_{cpu_2}	15	12	27
t_{cpu_3}	40	12	52

Tabelle 3: Laufzeiten und t_{delta}

Zeitstempel t_{base} = Zeitstempel von r_{base}
Zeitdifferenz $t_{delta} = t_{current} - t_{base}$
for all t_x in r_{base} **do**
$t_{interpolated} = t_x + t_{delta}$
end for
end if

Beispiel In der in Abbildung 20 zu sehenden Timeline sind zwei Records für die Zeitpunkte 60 und 84 enthalten. Gesucht ist ein Record für den Zeitpunkt $t_{current} = 72$. Da für diesen Zeitpunkt kein Record existiert, wird ein leerer Record $r_{interpolated}$ mit Zeitstempel 72 angelegt, der nächstkleinere Record r_{base} gesucht und bei $t_{base} = 60$ gefunden. Anschließend wird $t_{delta} = t_{current} - t_{base} = 72 - 60 = 12$ gesetzt. Als letzter Schritt werden die Zeiten für cpu_1, cpu_2 und cpu_3 in $r_{interpolated}$ eingetragen. Dadurch ergeben sich die Laufzeitwerte wie in Tabelle 3 abgebildet.

Da vor und nach jedem Methodenaufruf Messpunkte gesetzt sind, kann angenommen werden, dass sich die Zustände der Methoden nicht vom Basisdatensatz unterscheiden. Daher werden die Methodenzustände direkt vom Basisdatensatz in den interpolierten Datensatz kopiert.

Nachdem für beide Zeitpunkte ein Datensatz interpoliert wurde, wird der ältere Daten-

	r_α	r_Ω	$r_{intervall}$
t_{cpu_1}	34	22	12
t_{cpu_2}	39	27	12
t_{cpu_3}	64	52	12

Tabelle 4: Subtraktion zweier Datensätze

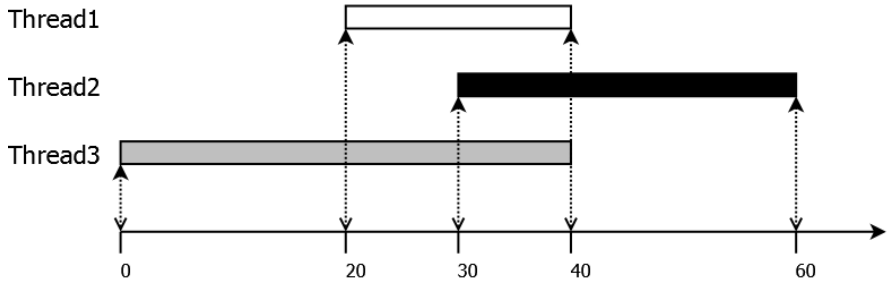


Abbildung 21: Beispiel für Laufzeiten auf einem logischen Prozessor CPU1

satz r_α vom jüngeren Datensatz r_Ω subtrahiert, also alle Zeiten und alle Methodenaufrufe innerhalb der Datensätze voneinander abgezogen. Daraus ergibt sich der gewünschte Datensatz $r_{intervall}$. Für den gegebenen Beispielintervall [72,84] wie in 20 ergeben sich die Laufzeitwerte wie in Tabelle 4 in der Spalte $r_{intervall}$ angegeben. Die Werte für die Anzahl der Methodenaufrufe ergeben sich analog.

5.6 Der Visualisierungsteil

Die Anzahl der Funktionen einer Visualisierung hat laut Sebrechts et al. [1999] einen höheren Stellenwert als die Anzahl ihrer Dimensionen. Dies deckt sich mit den Erkenntnissen von Shneidermann ([Shneiderman 1996]). Im Folgenden soll daher auf die implementierten Funktionen eingegangen werden.

5.6.1 Angezeigte Laufzeitdaten

Der *PlaybackController* kann die interpolierten Daten auf drei verschiedene Arten aufbereiten, *Absolute Runtime*, *Relative Runtime* und *Thread Spread*. Die verschiedenen Modi unterscheiden sich durch den Bezugspunkt, zu dem sie die hinterher angezeigten prozentualen Werte ermitteln. Abbildung 21 zeigt ein Beispiel für eine Ausführung von drei Threads auf einem logischen Prozessor. Dabei wird ein Intervall von 0 bis 60 betrachtet.

Der erste verfügbare Modus, Absolute Runtime, zeigt den prozentualen Anteil der Threadlaufzeit relativ zur Länge des gewählten Intervalls. Anhand dieser Werte kann die

	t_{thread}	$t_{duration}$	v
Thread1	20	60	30%
Thread2	30	60	50%
Thread3	40	60	75%

Tabelle 5: Prozentuale Werte für den Modus *Absolute Runtime*

Belastung, die ein Thread auf einen logischen Prozessor ausübt, näherungsweise abgelesen werden. Der angezeigte Wert v ergibt sich hier wie folgt:

Sei t_α Anfang des gewählten Intervalls, t_Ω Endzeit des Intervalls und $t_\alpha \neq t_\Omega$. Dann ist $t_{duration}$ gegeben durch $abs(t_\Omega - t_\alpha)$. Weiter sei t_{thread} die für diesen Intervall ermittelte Laufzeit für den gewählten Thread. Dann ist der angezeigte Wert v definiert als $v = (t_{thread}/t_{duration}) * 100$. Die Summe der innerhalb eines logischen Prozessors angezeigten Prozentwerte kann dabei 100% übersteigen, wenn zum Beispiel zwei Threads jeweils eine Laufzeit 60 Zeiteinheiten innerhalb eines 100 Zeiteinheiten fassenden Intervalls haben. Dies kann auftreten, da auch Zeiten, in denen ein Thread „schläft“, zur Laufzeit des Threads auf dem momentanen logischen Prozessor gerechnet werden. Schläft ein Thread, kann in dieser Zeit ein anderer Thread ausgeführt werden, dessen Laufzeit ebenfalls dem momentanen logischen Prozessor zugeschlagen wird. Diese Zeit wird also doppelt gebucht, wodurch mehr als 100% gesamte Laufzeit auftreten können. Dieses Problem ergibt sich aus dem in Kapitel 5.4.3 beschriebenen Problem. Für das Beispiel aus Abbildung 21 ergeben sich die in Tabelle 5 erfassten prozentualen Werte.

Der zweite Modus, Relative Runtime, berechnet den prozentualen Anteil an der *Summe aller Laufzeiten* innerhalb des gewählten Intervalls. Die in diesem Modus gelieferten Werte können z.B. genutzt werden, um zu bestimmen, ob ein bestimmter Thread im Vergleich zu anderen Threads mehr Arbeit zu leisten hat (und dadurch längere Laufzeiten erzielt), die eventuell auf andere oder neue Threads verteilt werden kann. Der angezeigte Wert v_i für einen Thread ist hier wie folgt definiert:

Sei $T_{runtimes} = t_1 ... t_n$ die Menge aller für diesen logischen Prozessor ermittelten Laufzeiten innerhalb des gewählten Intervalls. Dann ist $v_i = (t_i / \sum_{i \in 1...n} t_i) * 100$. Die Summe der innerhalb eines logischen Prozessors angezeigten Prozentangaben ergibt dabei stets 100% (abgesehen von kleineren Rundungsfehlern). Für das Beispiel aus Abbildung 21 ergeben sich die in Tabelle 6 erfassten prozentualen Werte.

Der dritte Modus, Thread Spread, zeigt die Verteilung eines Threads innerhalb des Intervalls. Die Summe der angezeigten Werte pro Thread ergibt hier stets 100%. In diesem Modus lässt sich beispielsweise prüfen, ob das Threadscheduling ausgeglichen ist. Eine gleichmäßige Verteilung auf allen oder einer festen Teilmenge von logischen Prozessoren deutet dabei auf eine gute Verteilung hin. Der angezeigte Wert v_{cpu_i} für den logischen Prozessor cpu_i ist folgendermaßen definiert:

	t_{thread}	$\sum_{i \in 1...n} t_i$	v_i
Thread1	20	90	22,22%
Thread2	30	90	33,33%
Thread3	40	90	44,44%

Tabelle 6: Prozentuale Werte für den Modus *Relative Runtime*

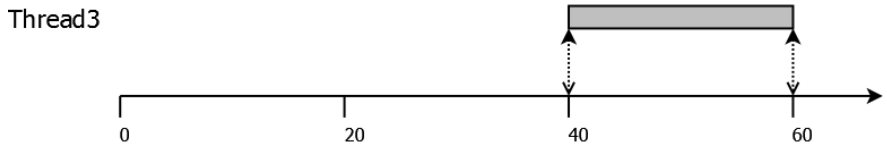


Abbildung 22: Beispiel für Laufzeiten auf einem logischen Prozessor CPU2

Sei $T_{cputimes} = t_{cpu_1}..t_{cpu_n}$ die Menge der für diesen Thread ermittelten Laufzeiten auf den logischen Prozessoren $cpu_1...cpu_n$. Dann ist $v_{cpu_i} = (t_{cpu_i} / \sum_{i \in 1...n} cpu_i) * 100$. Das Beispiel aus Abbildung 21 greift für diesen Modus nicht, da es nur einen einzelnen Prozessor betrachtet. Es würden sich für alle Werte 100% ergeben. Daher wird für diesen Modus das Beispiel um einen zweiten logischen Prozessor erweitert, dessen Verlauf in Abbildung 22 dargestellt ist. Dann ergeben sich für das Beispiel im Modus *Thread Spread* die in Tabelle 7 erfassten prozentualen Werte.

Die Visualisierung stellt zur Zeit die Laufzeit (*Wall Time*) der Threads dar. Der Begriff Wall Time leitet sich von der „Uhr *an der Wand*“ ab, also von der tatsächlichen, reellen Laufzeit. Die aktive Zeit eines Threads (*CPU Time*), also die Anzahl der Taktzyklen, die der Prozessor tatsächlich mit der Bearbeitung des Threads beschäftigt war, kann nicht dargestellt werden. Dies ergibt sich aus dem bereits angesprochenen Problem der Messungenauigkeit (Kapitel 5.4.3). Werden zwei oder mehr Threads innerhalb eines Intervalls parallel ausgeführt, so gibt es keine Möglichkeit zu bestimmen, ob und wo sie sich überlappen.

	t_{cpu_1}	t_{cpu_2}	v_{cpu_1}	v_{cpu_2}
Thread1	20	0	100%	0%
Thread2	30	0	100%	0%
Thread3	40	20	66,66%	33,33%

Tabelle 7: Prozentuale Werte für den Modus *Thread Spread*

5.6.2 Einschränkung des betrachteten Ausschnitts

Der Prototyp bietet die Möglichkeit, den betrachteten Ausschnitt nach Belieben einzugrenzen. Dazu werden, wie in Kapitel 5.5.2 beschrieben, Werte interpoliert und angezeigt. Dadurch kann eine beliebig fein aufgelöste Analyse der Daten durchgeführt werden.

5.6.3 Playback von Datensätzen

Der Prototyp bietet neben der Möglichkeit, eine Zusammenfassung eines gewählten Intervalls zu betrachten (siehe Kapitel 5.6.2), auch die Möglichkeit, die Veränderung des Systems über die Zeit zu betrachten. Die geschieht durch das Aneinanderreihen von Intervallen, wobei das Ende des aktuellen Intervalls den Beginn des nächsten Intervalls darstellt. Die Länge der Intervalle kann frei gewählt werden, die Geschwindigkeit des Playbacks ist festgelegt auf ca. eine Sekunde pro Intervall. Diese Einschränkung ergibt sich aus der Zeit, die die Visualisierung benötigt, um den Zustand der angezeigten Objekte flüssig darzustellen. Die Festlegung auf eine Sekunde entstand durch das Probieren verschiedener Geschwindigkeiten. Dabei hat sich gezeigt, dass ein zu schneller Wechsel zwischen den Zuständen hektisch wirkt. Der Benutzer hat nicht genügend Zeit, um Informationen aus den momentan angezeigten Daten zu ziehen, bevor das nächste Intervall angezeigt wird. Zu langsame Wechsel hingegen fordern die Geduld des Benutzers heraus und können bei bei großen Datenmengen schnell zu Ermüdung führen.

Aus diesen Gründen ist auch keine Wiedergabe in Echtzeit für den Prototypen implementiert oder geplant, obwohl das Kieker Framework diese Möglichkeit theoretisch liefern würde. Messpunkte liefern ggf. im Abstand von wenigen Millisekunden Daten, und Wechsel in derart kurzer Zeit sind für Menschen nicht erfassbar. Aus dieser Einschränkung erwächst jedoch ein praktischer Nutzen. Erst durch den Wegfall der Notwendigkeit, die eingehenden Daten in Echtzeit verarbeiten zu müssen, ist es möglich, die Daten vor der Anzeige wie in Kapitel 5.5 beschrieben aufzubereiten.

5.6.4 Navigation

Der Prototyp erlaubt die freie Bewegung im dreidimensionalen Raum. Um dem Benutzer Orientierungspunkte zu geben, ist der „Boden“ der Visualisierung mit einem Gitter versehen. Zusätzlich ist die Y-Achse im Ursprung dargestellt. Für eine schnelle Navigation innerhalb der Visualisierung können über Steuerelemente einzelne Blöcke direkt angesprungen werden. Ebenfalls über ein Steuerelement kann eine Übersichtsposition über die komplette Stadt angesprungen werden.

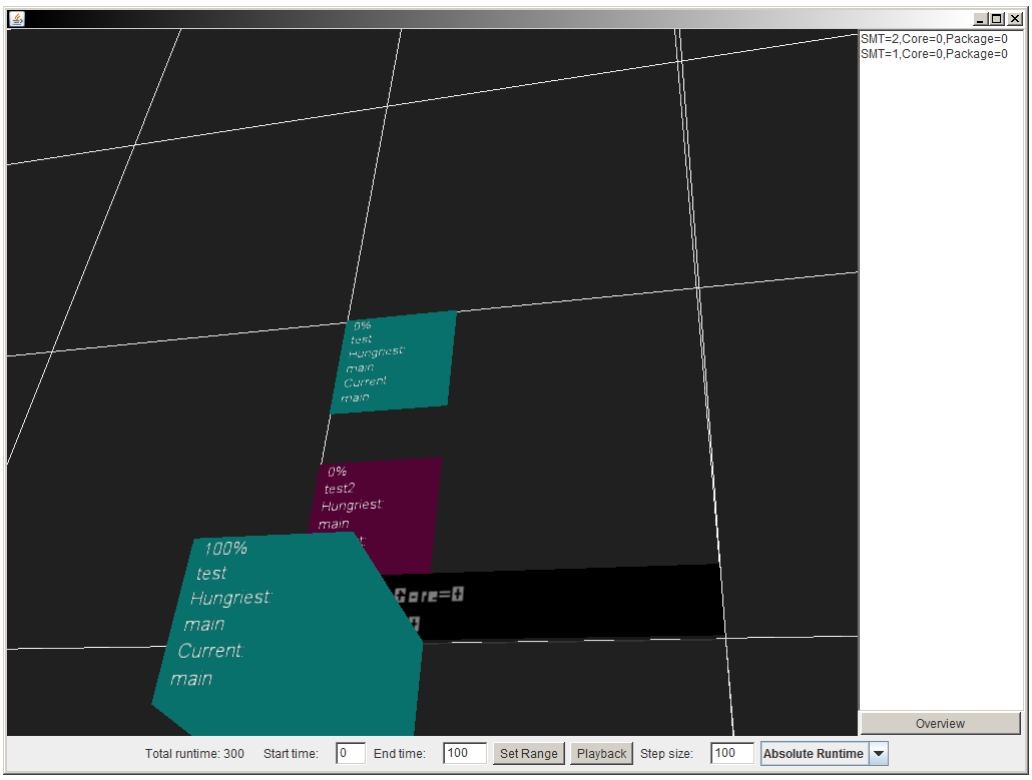


Abbildung 23: Screenshot des Prototypen

6 Evaluation

Nachdem im vorigen Kapitel einige Details und Besonderheiten der Implementierung beschrieben, deren Ergebnis in Abbildung 23 zu sehen ist. In diesem Kapitel folgt nun die Evaluation der erstellten Software. Zunächst erfolgt eine Analyse des Einflusses der erstellten Sonde auf den Betrieb der gemessenen Software. Dazu wird der Overhead der Sonde betrachtet.

Als nächstes wird der Prototyp auf die reine Funktionalität hin untersucht. Damit soll gezeigt werden, dass die vom Prototypen angezeigten Werte korrekt sind. Alle in diesem Kapitel beschriebenen Testszenarien befinden sich auf der beiliegenden CD.

Weiter erfolgt eine qualitative Evaluation des Prototypen anhand der in Kapitel 2.2 aufgeführten Kriterien. Damit sollen die weniger greifbaren Eigenschaften wie Bedienbarkeit und Übersichtlichkeit erfasst werden. Zusätzlich kann der Prototyp so mit den anderen, in Kapitel 3 beschriebenen Ansätzen, verglichen werden.

6.1 Evaluation des Overheads der Kieker Sonde

Das Kieker Framework ist ein Monitoring Werkzeug. Als solches sollte es einen möglichst kleinen Einfluss auf das überwachte System haben, um möglichst genaue Messwerte zu erhalten und den eigentlichen Programmablauf nicht zu behindern. Dies gilt besonders, wenn das Monitoring in Produktivsystemen eingesetzt werden soll. Eines der Ziele des Kieker-Frameworks ist daher das niedrig halten des Overheads, der durch die Sonden entsteht. Um den Overhead der für den Prototypen erstellten Sonde abschätzen zu können, wurde der Overhead der Methode ermittelt, die für die Sonde zusätzlich entwickelt wurde. Der Aufruf von CPUID durch das JNI wurde speziell für die Sonde des Prototypen hinzugefügt. Daher stellt sie potentiell einen zusätzlichen Overhead zu einer Standard Kieker Sonde dar. Weiterführende Informationen über den grundsätzlichen Overhead einer Kieker Sonde finden sich unter van Hoorn et al. [2012].

Um den Overhead durch CPUID und JNI zu ermitteln, wurden drei Testfälle erstellt und gemessen. Da für jeden der Testfälle die Dauer eines einzelnen Aufrufs weit unter einer Millisekunde liegt, wurde jeweils die Zeit für 10.000.000 Aufrufe gemessen. Um einen repräsentativen Mittelwert zu erhalten, wurde jeder Test 10.000x durchgeführt und der Mittelwert gebildet, während das Testsystem keine anderen Aufgaben hatte. Alle Testläufe wurden auf dem gleichen Testsystem durchgeführt. Das Testsystem besteht aus einer Intel Core i5-2410M@2.30 GHz CPU, mit 4GiByte RAM, auf dem ein Windows 7 Pro (64bit) installiert ist. Es wurde eine Oracle Java Version 1.7.0_01 (Client VM, 32 Bit) verwendet. Die Ergebnisse der Testläufe sind in Tabelle 8 abgebildet.

1) leerer JNI Aufruf Für diesen Testfall wurde eine leere C Methode ohne Rückgabewert über JNI aufgerufen. Die Messung der Zeit erfolgte um den JNI Aufruf herum, befindet sich also im Java-Part.

2) parametrierter leerer JNI Aufruf Dieser Testfall ist aufgebaut wie Testfall 1, jedoch wird der C Methode ein Integer als Parameter übergeben. Die C Methode gibt den Parameter unverändert wieder zurück. Dieser Testfall soll prüfen, ob zusätzliche Parameter einen Einfluss auf die Verarbeitung von JNI Aufrufen haben.

3) nativer Aufruf von CPUID (ohne JNI) Um den nativen Aufruf der CPUID Instruktion zu messen wurde eine C Methode implementiert, die in einer eigenen Schleife nur die CPUID Instruktion *CPUID_Fn0000_0000_EAX* aufruft. Die Zeitmessung wurde für je einen Schleifendurchlauf durchgeführt. Auf diese Weise kann die reine Aufrufzeit der CPUID Instruktion ermittelt werden, ohne zusätzlichen Overhead durch den Aufruf einer C Methode berücksichtigen zu müssen.

4) Aufruf CPUID mit JNI Dieser Testfall kommt dem tatsächlichen Aufruf innerhalb der Sonde am nächsten. Aus der Java Umgebung heraus wird einer C-Methode per JNI ein Integer Array übergeben. Das Array enthält die Parameter für den CPUID Aufruf. Nach dem Aufruf von CPUID wird, ebnfalls per JNI, ein zweites Integer Array an die Java Umgebung zurückgegeben. Dieses zweite Integer Array enthält die Rückgabewerte des CPUID Aufrufs. Die Zeitmessung wurde, wie in Testfall 1, um den JNI Aufruf herum.

Testfall	Durchschnittliche Laufzeit für 10.000.000 Aufrufe
1) Leerer JNI Aufruf	103 ms
2) Parametrierter leerer JNI Aufruf	109 ms
3) Nativer Aufruf von CPUID (ohne JNI)	362 ms
4) Aufruf CPUID mit JNI	2954 ms
5) Arraykonvertierung mit JNI	2730 ms

Tabelle 8: Testergebnisse für Overhead durch JNI und CPUID

5) Arraykonvertierung mit JNI Dieser Testfall ist aufgebaut wie Testfall 4, jedoch wird kein Aufruf zu CPUID getätigt. Stattdessen wird das übergebene Java Array in eine C Array umgewandelt und ein neues C Array in ein Java Array umgewandelt und zurückgegeben. Dieser Testfall soll den Overhead abdecken, der notwendig ist, um komplexere Java-Typen in C verwenden zu können.

Auffallend an diesen Ergebnissen ist der Unterschied der einzelnen Aufrufe in den Testfällen 2 und 3 zu Testfall 4. Anstatt einer erwarteten Laufzeit von ungefähr 109ms + 362ms = 371ms springt die Zeit auf 2954ms. Dieser Sprung ist mit der Konvertierung der Java Arrays in C Arrays und umgekehrt zu erklären, wie sich in dem Ergebnis für Testfall 5 zeigt.

Dennoch ist das Ergebnis von knapp drei Sekunden für zehn Millionen Aufrufe, also ca. 300ns pro Einzelaufruf, durchaus vertretbar für den Einsatz in einer zeitkritischen Umgebung wie dem Monitoring. Weitere Ergebnisse zum Overheades Kieker Frameworks werden in Waller and Hasselbring [2012] beschrieben.

6.2 Evaluation der Funktionalität des Prototypen

In diesem Abschnitt sollen die Tests und Ergebnisse beschrieben werden, mit denen die reine Funktionalität des Prototypen getestet wird.

6.2.1 Testaufbau Datenerfassung

Zunächst wird die Funktionalität der Sonde überprüft. Dazu wird eine Demoanwendung, der *JPetStore* ([iBatis 2009]), mit der entwickelten Sonde instrumentiert. Der JPetStore ist eine Apache-basierte Webanwendung, die einen interaktiven Tierkatalog skizziert. Da die Anwendung eine HTML Shopseite anbietet, können nun mithilfe von *JMeter* ([Apache]), einem Testwerkzeug für Webseiten und Webanwendungen, automatisch Aktionen auf der Seite ausgeführt werden. Die für diese Aktionen aufgerufenen Methoden werden von der Sonde gemessen und entsprechende Monitoring Records erzeugt.

JMeter benutzt zur Lasterzeugung einen zuvor aufgezeichneten Testplan. Der Testplan ist auf der beiliegenden CD enthalten. Der Testplan besteht aus dem Aufruf der JPetStore Seite, gefolgt von einem „Einkauf“. Dabei wird aus jeder angebotenen Tiergattung ein Exemplar in den Warenkorb gelegt und anschließend „bezahlt“. JMeter führt diesen Testplan automatisch mit 10 „Benutzern“ gleichzeitig durch. Dabei werden, wie erwartet, vom JPetStore zehn Threads gestartet, die sich in den Testdaten wiedererkennen lassen. Auffallend ist, dass während des Testlaufs auf dem Testsystem nicht alle 16 logischen Prozessoren für den JPetStore verwendet werden.

Der instrumentierte JPetStore wird auf einem Server mit zwei Intel Xeon E5540 Prozessoren und 24GB RAM ausgeführt. Jeder der Prozessoren verfügt über 4 Kerne und Hyperthreading. Es stehen also insgesamt 16 logische Prozessoren zur Verfügung. Als Betriebssystem kommt Solaris 10 zum Einsatz, auf der eine Oracle Java VM mit der Version 1.7.0_03 (Server, 64 Bit)installiert ist. Zur Lasterzeugung wird JMeter separat auf dem in Kapitel 6.1 beschriebenen Testsystem ausgeführt.

6.2.2 Testaufbau Datentransformation / Darstellung

Da die Funktionen der Darstellung stark von den aus dem Playback-Part gelieferten Daten abhängen, werden diese beiden Teile zusammen getestet. Dazu werden verschiedene Testszenarien in Form von Kieker Monitoring Records erstellt, die eingelesen, transformiert und dargestellt werden sollen. Jedes Szenario ist so gestaltet, dass es leicht überprüfbare Ergebnisse liefert. Das korrekte Funktionieren der Werkzeuge, wie CPUID oder des Kieker Frameworks, wird dabei vorausgesetzt.

Layout Test Der einfachste durchgeführte Test ist der Layout Test. Der Layout Test soll sicherstellen, dass alle Blocks und Häuser an der korrekten Stelle dargestellt werden, mit ausreichendem Abstand, um einzelne Objekte und Zugehörigkeiten zu unterscheiden. Zusätzlich soll der Farbwahlalgorithmus für die Häuser getestet werden.

Um dies zu überprüfen, werden Monitoring Records erstellt, die zur Darstellung von je 3 Threads auf 3 logischen Prozessoren führen. Die in den Records eingetragenen Zeitstempel sind dabei lediglich Dummies.

Das erwartete Ergebnis ist, dass auch nach wiederholtem Neustart des Prototypen stets 3 logische Prozessoren, also Blocks, mit je 3 Threads, also Häusern, dargestellt werden (siehe auch Abbildung 24). Alle angezeigten Objekte sollen sich klar unterscheiden lassen, es dürfen keine Überschneidungen in den Objekten existieren. Die Häuser sollen dabei stets die gleiche Farbe haben. Dies lässt sich beispielsweise durch Screenshots oder Werkzeuge wie ColorPicker ([ColorPic]) überprüfen.

Interpolationstest Dieser Test ist entworfen worden, um die Interpolationsroutinen des Prototypen zu testen. Dazu werden 6 Monitoring Records verwendet, die nur zwei Threads auf zwei logischen Prozessoren darstellen. Die verwendeten Records sind in Tabelle 9 aufgetragen. Anschließend wird der Prototyp gestartet und bestimmte Intervalle angewählt, für die die erwarteten Werte bekannt sind. Dieser Test wird für alle drei

Record	Zeitstempel	Threadname	Aktuelle CPU Id
Record 1	0	Thread_One	1
Record 2	100	Thread_One	2
Record 3	100	Thread_Two	2
Record 4	200	Thread_Two	1
Record 5	300	Thread_One	2

Tabelle 9: Monitoring Records für den Interpolationstest

Anzeigemodi, *Absolute Runtime*, *Relative Runtime* und *Thread Spread*, wiederholt. Die Tabellen 10, 11 und 12 listen die gewählten Intervalle und die erwarteten Werte auf.

Stress Test Das Stress Test Szenario ist konzipiert, um die Grenzen des Prototypen auszuloten und gezielt zu überschreiten. Dazu werden die verwendeten Monitoring Records schrittweise angepasst. Zunächst wird nur eine kleine Anzahl von Threads und logischen Prozessoren konfiguriert und der Prototyp gestartet. Bei erfolgreicher Ausführung wird die Anzahl von Threads und logischen Prozessoren erhöht und die Lauffähigkeit erneut überprüft. Dieser Vorgang wird wiederholt, bis der Prototyp nicht mehr Lauffähig ist. Bei einer Größe des Java Heap von 256 Megabyte kann der Prototyp je 128 Threads auf 12 logischen Prozessoren anzeigen. Eine Erhöhung auf je 128 Threads auf 13 logischen Prozessoren führt zur Unbenutzbarkeit des Programms, verursacht durch Speichermangel. Der Hauptgrund für diesen Speichermangel sind die 128x128 Pixel großen Texturen auf den Häuserdächern, auf denen die textuellen Informationen abgebildet sind. Bei 128 mal 13 Threads müssen mindestens 98 Megabyte Speicher nur für die Texturen zur Verfügung gestellt werden.

Eine Vergrößerung des Java Heaps löst dieses Problem. Werden der Java VM 1024 Megabyte Heap zur Verfügung gestellt, lassen sich weit mehr Threads darstellen Der Stress Test wurde bei 128 mal 24 Threads abgebrochen, da hier das Testsystem nicht mehr ausreichend flüssig läuft, um ein ernsthaftes Arbeiten zu ermöglichen.

Zusätzlich zu den synthetischen Szenarien wurden auch die unter Realbedingungen gesammelten Testdaten (siehe Kapitel 6.2.1) verwendet. Mit diesen Daten wurde die Visualisierung Stichpunktartig getestet.

6.2.3 Ergebnis der funktionalen Evaluation

Die funktionale Evaluation des Prototypen anhand der beschriebenen Tests hat ergeben, dass die reine Funktionalität des Prototypen gewährleistet ist.

	Thread_One		**Thread_Two**	
Intervall	CPU 1	CPU 2	CPU 1	CPU 2
0 - 50	100	0	0	0
50 - 150	50	50	0	50
250 - 300	0	100	100	0
0 - 300	33,33	66,67	33,33	33,33

Tabelle 10: Testergebnisse für den Interpolationstest (Absolute Runtime Modus)

	Thread_One		**Thread_Two**	
Intervall	CPU 1	CPU 2	CPU 1	CPU 2
0 - 50	100	0	0	0
50 - 150	100	50	0	50
250 - 300	0	100	100	0
0 - 300	50	66,67	50	33,33

Tabelle 11: Testergebnisse für den Interpolationstest (Relative Runtime Modus)

	Thread_One		**Thread_Two**	
Intervall	CPU 1	CPU 2	CPU 1	CPU 2
0 - 50	100	0	0	0
50 - 150	50	50	0	100
250 - 300	0	100	100	0
0 - 300	33,33	66,67	50	50

Tabelle 12: Testergebnisse für den Interpolationstest (Thread Spread Modus)

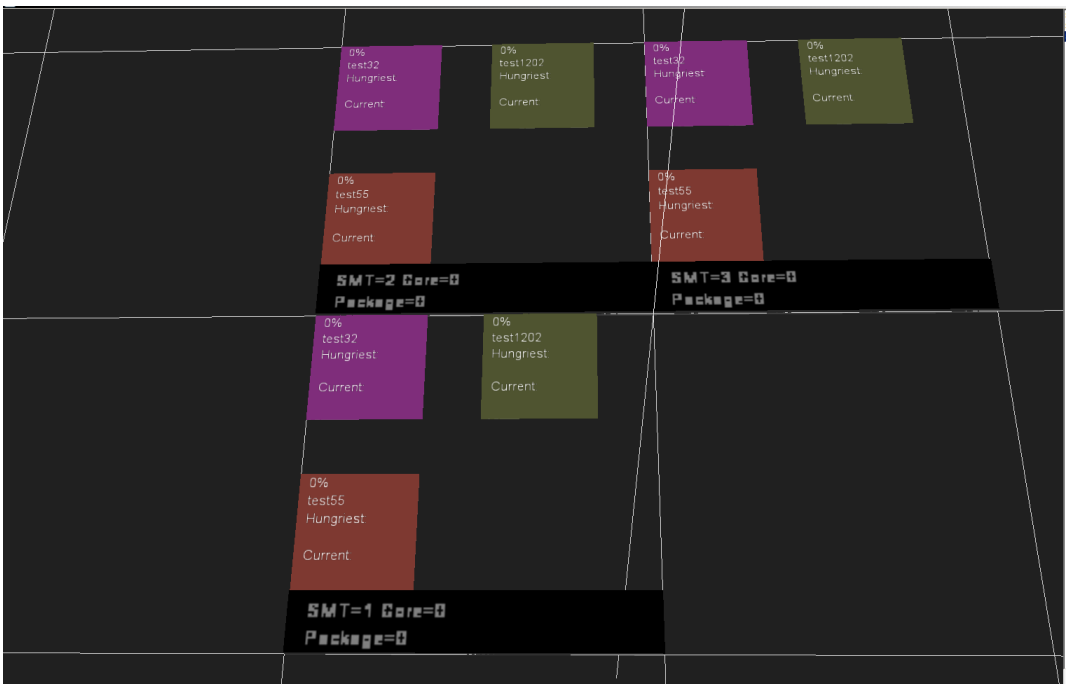


Abbildung 24: Screenshot des Layout Tests

6.3 Evaluation der Konzeptanforderungen

In diesem Abschnitt sollen die Anforderungen, die in Kapitel 4 aus den aufgestellten Kriterien abgeleitet wurden, evaluiert werden.

Visuelles Design Das Design ist weitestgehend wie in Kapitel 4.1 beschrieben umgesetzt. Im Laufe der Entwicklung ergaben sich jedoch Änderungen am Layout und der Aufteilung der einzelnen Ebenen. Es zeigte sich, dass es sinnvoller ist, sich auf die Beziehung einzelner Threads zu einem logischen Prozessor zu konzentrieren, statt auf die hierarchische Ordnung des Multiprozessorsystems. Daher wird die hierarchische Aufteilung aufgegeben und jeder logische Prozessor einzeln dargestellt. Da nun eine räumliche Trennung zwischen einzelnen logischen Prozessoren besteht, kann auch die Vereinheitlichung der Farbe innerhalb eines logischen Prozessors aufgegeben werden. Stattdessen wird die in Kapitel 5.2.2 beschriebene Lösung gewählt, bei der eine Farbkodierung der Threads statt der logischen Prozessoren stattfindet.

Kamera und Bedienung Die Anforderung, feste Punkte anspringen zu können, wird durch die Möglichkeit, alle angezeigten Blocks per Mausklick anspringen zu können, erfüllt. Jeder Block kann durch einen Doppelklick auf die Liste der Blocks rechts neben der Stadtanzeige ausgewählt werden.

Funktion	Taste
→	Schwenk rechts
←	Schwenk links
↑	Vorwärts
↓	Rückwärts
BildAuf	Schwenk runter
BildAb	Schwenk hoch

Tabelle 13: Tastenbelegung der Kamerasteuerung

Zudem ist die Kamera mithilfe der Tastatur frei dreh-, schwenk- und positionierbar. Die Tastenbelegung ist jedoch nicht frei wählbar. Die Tastenbelegung für die Kamerasteuerung ist in Tabelle 13 dargestellt. Auf eine optionale Maussteuerung wurde im Rahmen des Prototypen verzichtet.

Datenerfassung und abgeleitete Informationen Alle Anforderungen an diesen Bereich werden vom Prototypen erfüllt. Die Sonde ist in der Lage, die Id des momentanen logischen Prozessors, den Namen des momentanen Threads, den Namen der aktuellen Methode sowie einen aktuellen Zeitstempel auszulesen.

Durch die Verwendung von Java und des Kieker Frameworks ist die Plattformunabhängigkeit weitgehend erhalten geblieben. Lediglich der native Aufruf der CPUID Instruktion muss für jede Plattform einzeln kompiliert werden,

Implementierung / Architketur Die Architektur ist wie in Kapitel 5.3 umgesetzt. Sie behält die im Vorfeld geforderte Dreiteilung des Prototypen in Datenerfassung, Datentransformation und Darstellung bei.

6.3.1 Ergebnis der Evaluation

Der Prototyp erfüllt, mit den genannten Ausnahmen, alle im Vorfeld gestellten Anforderungen. Lediglich im Bereich der tatsächlichen Darstellung wurde von der Planung abgewichen, um auf neue Erkenntnisse zu reagieren.

6.4 Qualitative Evaluation des Prototypen

Nachdem im vorigen Kapitel die Funktionalität des Prototypen evaluiert wurde, soll in diesem Abschnitt die Evaluation der weniger greifbaren Eigenschaften erfolgen. Dazu gehören Eigenschaften, deren Qualität nicht durch bloßes Vorhandensein oder durch das Messen bestimmter Werte erfassbar sind. Dazu werden zum Einen die eigens dafür aufgestellten Kriterien aus Kapitel 2.2 herangezogen, zum Anderen eine kurze Übersicht über die bekannten Stärken und Schwächen des Prototyps gegeben.

6.4.1 Evaluation anhand der aufgestellten Kriterien

Analog zu den in Kapitel 3 vorgestellten Visualisierungstechniken soll der Prototyp anhand der in Kapitel 2.2 entwickelten Kriterien evaluiert werden.

Vollständigkeit Das Kriterium der Vollständigkeit (siehe Kapitel 2.2.1) ist anhand der Daten definiert, die von der Kieker Sonde geliefert werden. Die Visualisierung wurde so konzeptioniert und implementiert, dass alle geforderten Kerndaten angezeigt werden können. Dabei erfüllt der Prototyp die Anforderungen an die harte Auslegung der Vollständigkeit, da alle Kerndaten innerhalb der Darstellung abgebildet werden können. In Bezug auf die semantische Vollständigkeit zeigt der Prototyp explizit die Methode mit den meisten Aufrufen im gewählten Intervall. Mit dem Darstellungsmodus *Thread Spread* kann die Verteilung eines Threads auf die vorhandenen logischen Prozessoren angezeigt werden. Dies stellt einen weiteren Beitrag zur semantischen Vollständigkeit dar.

Granularität Die Granularität des Prototypen lässt sich hauptsächlich über die Wahl des darzustellenden Zeitraums steuern. Dabei wird durch die Interpolation neuer Messpunkten aus bestehenden Messpunkten die zeitliche Granularität verbessert. Durch das Springen zu einzelnen logischen Prozessoren ist es in begrenztem Maße auch möglich, die Menge der dargestellten Objekte einzuschränken. Ein tatsächliches Ausblenden von Objekten fehlt.

Räumliche Aufteilung / Orientierung Um eine übersichtliche Aufteilung im Sinne dieses Kriteriums zu erhalten, grenzt der Prototyp auf oberster Ebene die logischen Prozessoren klar voneinander ab. Innerhalb eines logischen Prozessors wird die Wiedererkennbarkeit eines einzelnen Threads durch die Beibehaltung der Position und Farbe einzelner Threads gewährleistet. Durch die Möglichkeit, per Doppelklick in der rechts neben der Darstellung angeordneten Liste zu einem bestimmten logischen Prozessor zu springen, wird dem Benutzer zusätzlich die Orientierung erleichtert. Durch den Layout-Algorithmus, der für die logischen Prozessoren verwendet wird, fehlt jedoch eine Gruppierung von logischen Prozessoren mit gleicher Core Id oder Package ID.

Animation Der Prototyp ist so implementiert, dass möglichst wenige Bewegungen und Veränderungen der Objekte notwendig sind. Daher ist die einzige Animation in der Visualisierung die Höhenänderung der Häuser in Abhängigkeit zu der Laufzeit der Threads, denen sie zugeordnet sind. Da die textuellen Darstellungen, also die auf dem Dach der Häuser angezeigten Informationen, ohne Veränderung der Position auf dem Dach des Hauses verändert werden, wird hier auf ein Überblenden der Werte verzichtet. Die Anzeige der Blocknamen wird während des Programmablaufs nicht mehr verändert und daher auch nicht animiert.

Aufschlüsselung der Informationen Der Prototyp stellt den größten Teil seiner Informationen textuell dar. Diese Informationen sind für den Benutzer leicht zu interpretieren. Im Falle von Unklarheiten, wie der Unterscheidung zwischen *Hungriest Method* und momentan ausgeführter Methode, ist der angezeigte Wert deutlich durch eine zusätzliche Beschriftung markiert. Durch die Beibehaltung von Threadfarben, auch über mehrere

Programmläufe hinweg, wird dem Benutzer das Erkennen der durch ihn beobachteten Threads erleichtert. Die Häuser als Metapher für einzelne Threads sind leicht nachvollziehbar. Die Änderungen in der Höhe der einzelnen Häuser als Repräsentation des aktuellen Wertes wird zusätzlich durch die textuelle Darstellung auf dem Dach gestützt. Die ComboBox zur Auswahl des Darstellungsmodus ist jederzeit sichtbar, so dass die Interpretation des dargestellten Wertes jederzeit möglich ist.

6.4.2 Stärken

Die größte Stärke des Prototypen ist die Möglichkeit, den betrachteten Zeitabschnitt frei zu wählen. Dies ist durch die in Kapitel 5.5.2 beschriebene Interpolation in drei verschiedenen Modi möglich. Im Bereich der eigentlichen Visualisierung ist das Ermitteln der Farbe eines Hauses anhand des Threadnamens erwähnenswert. Dadurch wird die Verfolgbarkeit eines Threads über mehrere Programmläufe hinweg erleichtert. Die kann nützlich sein, wenn Software während ihrer Entwicklung in kurzen Intervallen analysiert wird. Diese Eigenschaften lassen sich jedoch auf nahezu alle der zuvor vorgestellten Visualisierungstechniken anwenden, so dass der implementierte Prototyp hier keinen Vorteil bietet.

6.4.3 Schwächen

Ebenso wie die größte Stärke des Prototypen, die Interpolation von Messpunkten, liegt die größte Schwäche des Prototypen nicht in der tatsächlichen Visualisierung, sondern in der Bereitstellung der Daten. Dem Prototypen fehlt eine Möglichkeit, tatsächlich parallel laufende Threads zu erkennen (siehe Kapitel 5.6.1). Daher sind alle angezeigten Werte lediglich zur groben Analyse der Software geeignet.

Eine Schwäche in der Visualisierung ist der Übersichtsverlust bei hoher Threadzahl (siehe Kapitel 5.2.1). Das gewählte Layout und die ständige Anzeige aller Threads ist ein Kompromiss zu ständigen großen Veränderungen des Layouts. Diese Schwäche ist auch durch die Wahl der City Metapher bedingt, die eher für die Darstellung von Daten geeignet ist, die, wie eine echte Stadt, nur geringen Änderungen über die Zeit unterliegen.

Eine Schwäche der Implementierung ist die hohe Last, die Java3D auf dem Testsystem erzeugte. Eine einfache Anzeige, ohne aktive Animation, erzeugte bereits circa 50% Auslastung beider Prozessorkerne.

6.4.4 Ergebnis der qualitativen Evaluation

Die eigentliche Darstellung erfüllt alle in Kapitel 2.2 aufgestellten Kriterien. Der Prototyp ist in einem begrenzten Rahmen zu Analyse von Softwaresystem geeignet, stellt jedoch hauptsächlich eine Basis dar, auf der neue, verfeinerte Ansätze aufbauen können.

Die Darstellung in Form der City Metapher bietet in dem gegebenen Kontext keinen Vorteil gegenüber zweidimensionalen Darstellungen wie z.B. einer Heatmap oder

eines Balkendiagramms. Der Grund dafür ist, dass nicht genug verschiedene Informationen dargestellt werden müssen, die sich graphisch darstellen lassen. Dadurch werden die Möglichkeiten der City Metapher und der 3D-Darstellung nicht vollständig genutzt.

7 Ausblick

Nachdem in den vorangegangenen Kapiteln bereits vorhandene Arbeiten und der neu geschaffene Prototyp vorgestellt wurden, soll an dieser Stelle ein Ausblick auf mögliche, weiterführende Arbeiten gegeben werden. Das Hauptaugenmerk liegt dabei auf der Erweiterung und Verfeinerung des erstellten Prototypen. Im Laufe der Entwicklung und Evaluierung ergaben sich einige Punkte, an denen der Prototyp verbessert werden kann. Diese Punkte werden in den folgenden Abschnitten beschrieben.

7.1 Erweiterung um zusätzliche Prozessortypen

Die Sonde des Prototypen funktioniert zur Zeit nur mit bestimmten Intel-Prozessoren (siehe auch Kapitel 5.4.4). Eine naheliegende Folgearbeit wäre, die erstellte C-Bibliothek um die CPUID-Aufrufe für andere und ältere Prozessormodelle zu erweitern. Dazu müsste für jeden zu unterstützenden Prozessor der korrekte CPUID-Aufruf implementiert werden. Die Vorgehensweise zum Auslesen der Id eine es logischen Prozessors findet sich oftmals in den Handbüchern der Hersteller, siehe auch [AMD] und [Intel].

7.2 Erhöhung der Messgenauigkeit

Wie in Kapitel 5.4.3 beschrieben, können Lücken in der Datenerfassung der Sonde auftreten. Dies ist bedingt durch die Art und Weise der Messung. Die Messpunkte der Sonde, am Anfang und am Ende einer Methode, sind nicht abhängig von den auftretenden Threadwechseln. Ein Punkt für die Erweiterung ist daher, eine Möglichkeit zu finden, die auftretenden Threadwechsel zu erfassen. Dazu müssten Nachrichten des Betriebsystem-Schedulers über einen Threadwechsel abgefangen und ausgewertet werden. Diese Methode kann jedoch zum Verlust der Plattformunabhängigkeit führen.

7.3 Erweitern der Methodenauswertung

Zur Bestimmung der *Hungriest Method* wird zur Zeit die Anzahl der Aufrufe einer Methode herangezogen. Eine Alternative dazu wäre, die Laufzeit der Methode als Vergleichsfaktor zu verwenden. Die Herausforderung liegt hierbei in der Erkennung geschachtelter oder rekursiver Aufrufe. Ohne eine entsprechende Behandlung wäre die *Hungriest Method* ansonsten stets die *main*-Methode, da sie ja von Anfang bis Ende der Anwendung lief. Um Verschachtelung und Rekursion zu erkennen, müsste die Sonde einen kompletten Stacktrace aufbauen, der später ausgewertet werden kann.

7.4 Erweiterte Bedienung des Prototypen

Der Prototyp bietet zur Zeit eine einfache, funktionale Steuerung. Um den Komfort bei der Bedienung zu erhöhen, könnten zusätzliche Bedienmöglichkeiten geschaffen werden.

Dazu könnten die Steuerung mit der Maus oder eine frei konfigurierbare Tastaturbelegung gehören. Weiterführung dieses Gedankens ist, die Steuerung völlig frei konfigurierbar für Maus und Tastatur zu gestalten. Eine zusätzliche Anforderung wäre dann, sinnvolle Voreinstellungen zu finden.

Auch die Anbindung neuer oder exotischer Eingabegeräte wie 3D-Mäusen oder Bewegungs- und Gestensteuerung kann untersucht werden.

7.5 Frei konfigurierbare Anzeigen bzw. Beschriftungen

Der Prototyp könnte dahingehend erweitert werden, das die auf dem Dach der Häuser dargestellten Informationen vom Anwender frei definiert werden können. Dadurch wird dem Anwender zusätzliche Kontrolle gegeben. Diese Erweiterung dient zur Erhöhung der *Granularität* des Prototypen.

8 Fazit

In der vorliegenden Arbeit wurde erforscht, wie die Prozessorauslastung innerhalb eines Multiprozessorsystems dargestellt werden kann. Dabei wurde der Ansatz einer dreidimensionalen Darstellung, einer City Metapher, gewählt, da die City Metapher eine große Flexibilität bei der Darstellung von Informationen bietet. Es galt zu ermitteln, was für die Entwicklung eines Prototypen zu beachten ist und was aus dem Prototypen selbst für Erkenntnisse gewonnen werden können.

Dazu wurden zunächst Arbeiten zur Bewertung und Einordnung von Visualisierungstechniken ausgewertet. Weiter wurden verschiedene Visualisierungstechniken untersucht, um einen Überblick über den aktuellen Stand zu erhalten. Dadurch wurde eine Grundlage für die weitere Arbeit geschaffen. Aus dieser Grundlage wurden eigene Kriterien entwickelt, die den gegebenen Kontext der Prozessorauslastung berücksichtigen. Anhand der entwickelten Kriterien wurden konkrete Anforderungen an einen zu implementierenden Prototypen abgeleitet. Daraufhin wurde der Prototyp entsprechend der Anforderungen entwickelt und implementiert. Der fertige Prototyp wurde dann gegen sowohl gegen die erstellten Kriterien für Visualisierungstechniken als auch gegen die konkreten Anforderungen geprüft. Abschließend wurden Anregungen zur Erweiterung und Verbesserung des Prototypen gegeben, basierend auf den Erkenntnissen, die während der Entwicklung und Implementierung gewonnen wurden.

Ergebnisse der Arbeit

Das Ergebnis der Arbeit ist zum einen eine Menge von Kriterien und Anforderungen, an denen sich zukünftige Arbeiten mit ähnlichen Zielsetzungen orientieren können, zum anderen ein konkretes Werkzeug. Dieses Werkzeug kann, mit den durch den Prototypstatus bedingten Einschränkungen, zur Analyse der Kernauslastung innerhalb eines Multiprozessorsystems eingesetzt werden.

Eine wichtige Erkenntnis der Arbeit ist, das die Nützlichkeit des erstellten Prototypen stark von den verfügbaren Daten abhängt. Die Interpolation von fehlenden Datenpunkten und die fehlende Möglichkeit, Threadwechsel genau zu erfassen, haben einen großen Einfluss darauf, welche Schlüsse der Anwender aus dem Werkzeug ziehen kann. Daher ist es angeraten, sich in Erweiterungen und Ergänzungen zum Prototypen auf die Datenerfassung zu konzentrieren.

Die abschließende Erkenntnis der Arbeit ist, dass die Flexibilität der City Metapher nicht ausgereizt wurde. Beispielsweise enthält die Fläche, die die einzelnen Häuser oder Blöcke belegen, keine Informationen. Die dargestellten Informationen können also auch in einer weniger aufwendigen Form präsentiert werden, beispielsweise in einer Heatmap oder einem einfachen 2D-Balkendiagramm, solange die Visualisierung keine weiteren Informationen anzeigen soll. Kommen jedoch durch zukünftige Erweiterungen weitere Informationen hinzu, können diese einfach in eine City Metapher eingefügt werden.

9 Anhang

9.1 Inhalt der CD

Die dieser Arbeit beiliegende CD enthält

- eine elektronische Fassung dieser Arbeit als PDF Dokument (Diplomarbeit.pdf)
- eine Version des implementierten Prototypen (ProCity.jar)
- eine Version der implementierten Sonde (ProcityProbe.jar)
- eine 32-Bit Windows Library für den Aufruf von CPUID (cpuid.dll)
- eine 64-Bit Solaris Library für den Aufruf von CPUID (libcpuid.so)
- die Java3D Bibliotheken (Ordner *workspace/lib*)
- den Quelltext des implementierten Prototypen (Ordner *workspace*)
- den Quelltext für die erstellte C-Bibliothek (cpuid.c)
- die im Laufe der Entwicklung generierten Testdaten (Ordner *testData*)
- den für JMeter verwendeten Testplan (Aggregate Report.jmx)
- eine Sammlung der Literatur, auf die in der Arbeit verwiesen wird (soweit Verfügbar) (Ordner *Literatur*)

9.2 Instrumentierung

Um eine Anwendung mit der Sonde zu instrumentieren, muss die der Pfad zu der Datei cpuid.dll (bzw. libcpuid.so) in den Java Library Path aufgenommen werden. zusätzlich müssen ProcityProbe.jar, kieker-1.4.jar, aspectjweaver-1.6.11.jar und commons-logging-1.1.1.jar im Classpath aufgeführt werden. Weiter muss eine aop.xml (siehe Listing 10) in einem META-INF Ordner zur Verfügung stehen. Die Anwendung kann dann mit einem Aufruf gestartet von der Kommandozeile aus gestartet werden.

Beispiel

```
java -javaagent:lib\aspectjweaver-1.6.11.jar -Djava.library.path=lib\
 -XX:-UseSplitVerifier
 -classpath lib\kieker-1.4.jar;lib\commons-logging-1.1.1.jar;
        lib\aspectjweaver-1.6.11.jar;lib\ProcityProbe.jar
 bookstoreTracing.BookstoreStarter
```

Listing 9: Beispiel zum Aufruf einer instrumentierten Anwendung

Dieser Aufruf startet die *Main* Methode der Klasse *bookstoreTracing.BookstoreStarter*. Die gesammelten Daten finden sich nach dem Programmdurchlauf im *%Temp%* Ordner des Systems. Beim diesem Beispiel wird angenommen, dass alle für die Sonde benötigten Dateien in einem Unterverzeichnis *lib* liegen.

```
<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN"
        "http://www.aspectj.org/dtd/aspectj_1_5_0.dtd">

<aspectj>
    <weaver options="">
        <include within="bookstoreTracing.BookstoreStarter"/>
    </weaver>

    <aspects>
      <aspect name="kieker.monitoring.probe.aspectJ.ProcityProbe"/>
    </aspects>
</aspectj>
```

Listing 10: AOP.xml Beispiel

9.3 Bedienung des Prototypen

Der Prototyp kann über die Kommandozeile gestartet werden. Dabei werden die Pfade zu den auszuwertenden Monitoring Records als Parameter übergeben. Zusätzlich müssen die nativen Java3D Bibliotheken (32 Bit oder 64 Bit) über den Java Library Path zur Verfügung gestellt werden. Das Beispiel 11 geht davon aus, das die notwendigen DLLs im Verzeichnis *lib* liegen.

```
java -Djava.library.path=\lib\ -jar Procity.jar Pfad/zu/den/Monitoring/Records/
```

Listing 11: Aufruf des Prototypen

Overview Panel

Im Overview Panel (siehe Abbildung 25) werden alle dargestellten logischen Prozessoren aufgelistet. Per Doppelklick auf den entsprechenden logischen Prozessor kann der entsprechende Block direkt aufgerufen werden. Mit dem *Overview* Button kann eine Übersicht über die gesamte Stadt angezeigt werden.

Control Panel

Das Control Panel enthält die grundlegenden Bedienelemente des Prototyps. Die Aufschlüsselung der in Abbildung 25 nummerierten Bedienelemente ist wie folgt:
1) *Total runtime* Label Dieses Label zeigt die Laufzeit des gesamten Testlaufs in generischen Zeiteinheiten an.

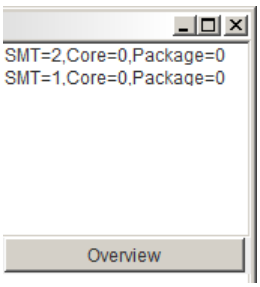


Abbildung 25: Overview Panel

2) *Start time* Eingabefeld Hier kann der Startzeitpunkt eines zu betrachtenden Intervalls eingetragen werden.
3) *End time* Eingabefeld Hier kann der Endzeitpunkt eines zu betrachtenden Intervalls eingetragen werden.
4) *Set Range* Button Beim Betätigen dieses Buttons wird der mit den Controls 2 und 3 definierte Intervall angezeigt.
5) *Playback* Button Der Playback Button startet die Animation der Darstellung mit dem in Control 2 eingestellten Startzeitpunkt und der in Control 6 eingestellten Schrittweite.
6) *Step size* Eingabefeld In diesem Eingabefeld kann die Schrittweite für die Animation eingetragen werden.
7) Moduswahl Mit dieser Combobox kann zwischen den in Kapitel 5.6.1 beschriebenen Anzeigemodi *Absolute Runtime*, *Relative Runtime* und *Thread Spread* umgeschaltet werden.

Abbildung 26: Control Panel

9.4 Klassendiagramme

Aus Platzgründen sind die Klassendiagramme, auf die im Kapitel 5.3 verwiesen wird, hier gesammelt.

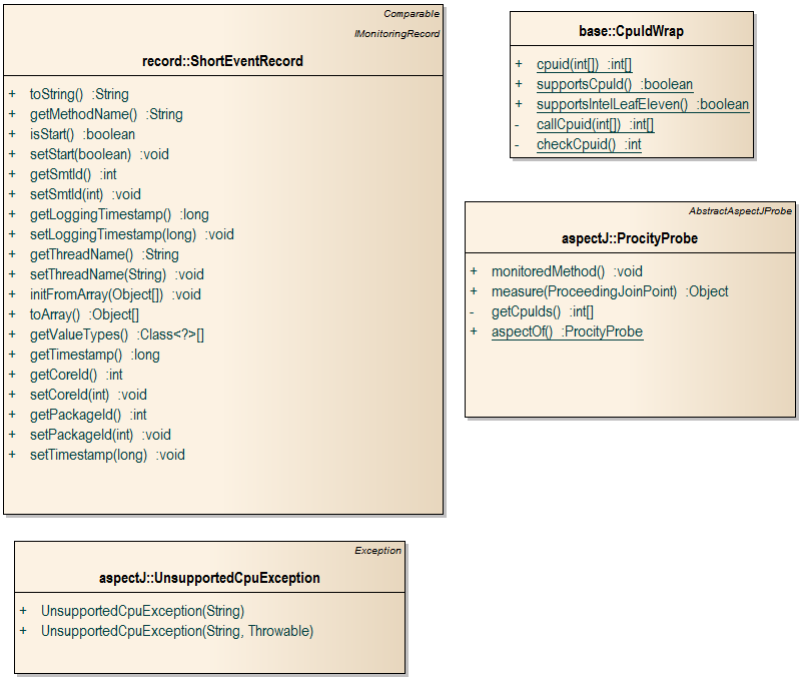


Abbildung 27: Klassendiagramm der Datenerfassung

Abbildung 28: Klassendiagramm der Datenauswertung

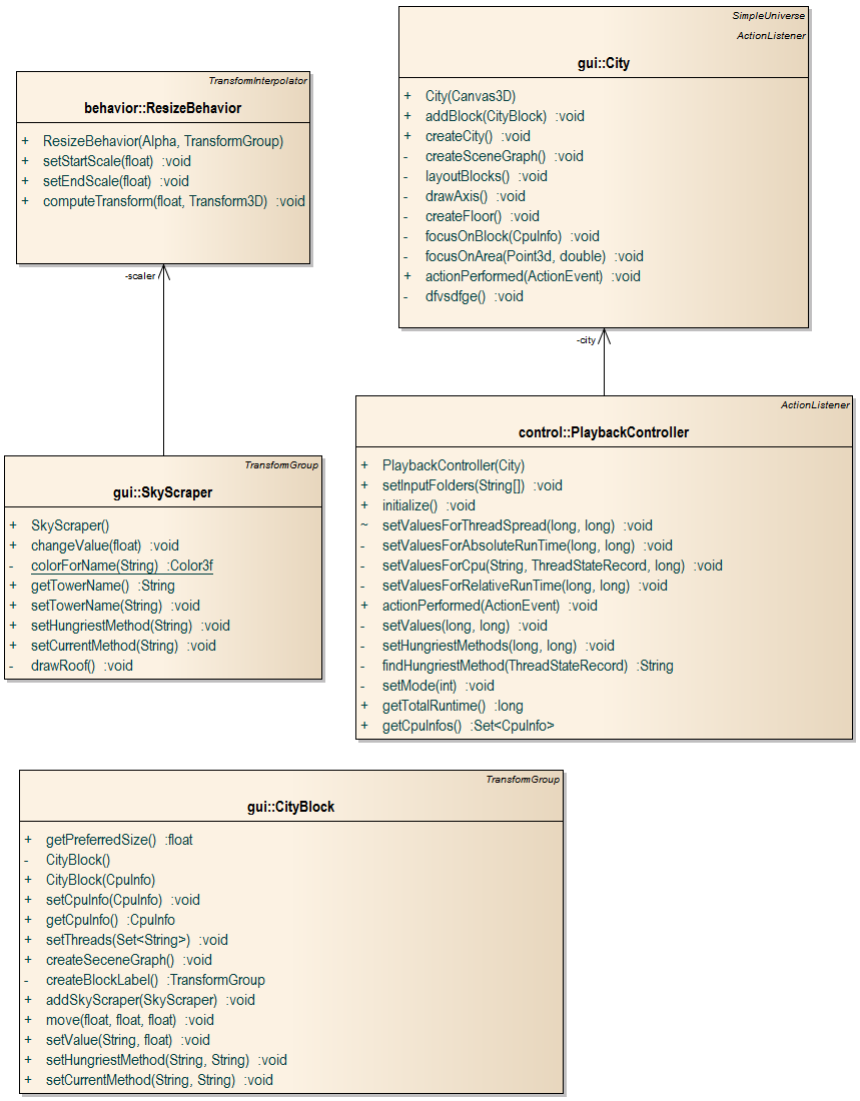


Abbildung 29: Klassendiagramm der Darstellung (1)

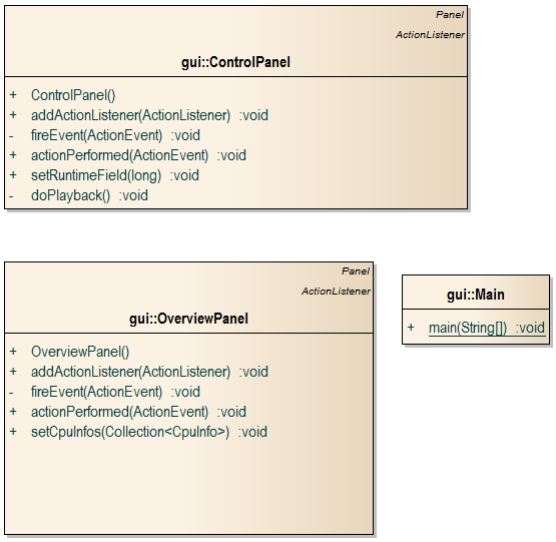


Abbildung 30: Klassendiagramm der Darstellung (2)

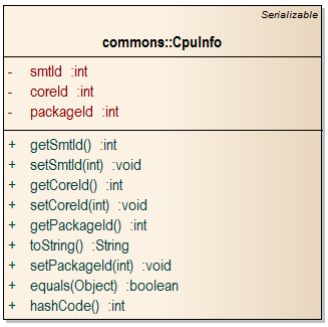


Abbildung 31: Klassen im commons Paket

Literatur

[j3d 1999] *Java 3D API Specification*, June 1999. Abgerufen 23.01.2012.

[AMD] AMD. *CPUID Specification*. Advanced Micro Devices, Inc, September 2010.

[Alam and Dugerdil 2007] S. Alam and P. Dugerdil. Evospaces visualization tool: Exploring software architecture in 3d. In *14th Working Conference on Reverse Engineering (WCRE).*, pages 269–270. IEEE, 2007.

[Apache] Apache. Apache JMeter. `http://jmeter.apache.org/`, 2012. Abgerufen 20.05.2012.

[Chafik 2012] O. Chafik. bridj. `http://code.google.com/p/bridj/`, 2012. Abgerufen 18.04.2012.

[Chi 2000] E. Chi. A taxonomy of visualization techniques using the data state reference model. In *IEEE Symposium on Information Visualization (InfoVis)*, pages 69–75. IEEE, 2000.

[ColorPic] ColorPic. ColorPic. `http://www.iconico.com/colorpic/`, 2012. Abgerufen 20.05.2012.

[Couch et al. 2012] J. Couch et al. Aviatrix3D. `http://aviatrix3d.j3d.org/`, 2012. Abgerufen 23.04.2012.

[Freitas et al. 2002] C. Freitas, P. Luzzardi, R. Cava, M. Winckler, M. Pimenta, and L. Nedel. On evaluating information visualization techniques. In *Proceedings of the working conference on Advanced Visual Interfaces*, pages 373–374. ACM, 2002.

[Fuhrmann and Gröller 1998] A. Fuhrmann and E. Gröller. Real-Time Techniques for 3D Flow Visualization, 1998.

[Heath and Finger 2003] M. Heath and J. Finger. Paragraph: A performance visualization tool for MPI. *URL: http://www.csar.uiuc.edu/software/paragraph/userguide.pdf*, August 2003.

[Hopkins 2001] G. Hopkins. The Joy of Java 3D. `http://www.java3d.org/starting.html`, 2001. Abgerufen 23.04.2012.

[Hruska 2011] J. Hruska. Win 8 Task Manager Now Heat Maps CPU Usage. `http://hothardware.com/News/Win-8-Task-Manager-Now-Heat-Maps-CPU-Usage/`, 10 2011. Abgerufen 14.01.2012.

[Hruska 2012] J. Hruska. The death of CPU scaling: From one core to many — and why we're still stuck. `http://www.extremetech.com/computing/116561-the-death-of-cpu-scaling-from-one-core-to-many-and-why-were-still-stuck`, February 2012. Abgerufen 28.05.2012.

[iBatis 2009] iBatis. JPetStore. `http://sourceforge.net/projects/ibatisjpetstore/`, 2009. Abgerufen 20.05.2012.

[Intel] Intel. *Intel Processor Identification and the CPUID Instruction*. Intel Corporation, January 2011.

[JAVA97] JAVA97. What is the Java Platform? `http://java.sun.com/docs/white/platform/javaplatform.doc1.html`, 1997. Abgerufen 23.04.2012.

[JNA] JNA. Java Native Access. `https://github.com/twall/jna`, 2012. Abgerufen 18.04.2012.

[JOGL] JOGL. JOGL. `http://jogamp.org/jogl/www/`, 2012. Abgerufen 23.04.2012.

[Kieker Project 2012a] Kieker Project. Kieker web site, 2012a. URL `http://kieker-monitoring.net/`.

[Kieker Project 2012b] Kieker Project. *Kieker 1.5 User Guide*. Software Engineering Group, University of Kiel, Germany, Apr. 2012b. URL `http://kieker-monitoring.net/documentation/`.

[Kuo 2009] S. Kuo. *Intel® 64 Architecture Processor Topology Enumeration*. Intel Corporation, December 2009.

[Lauxtermann 1995] F. Lauxtermann. Profiling paralleler Programme mit ParaGraph, Juli 1995.

[Liang 1999] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.

[Pastizzo et al. 2002] M. Pastizzo, R. Erbacher, and L. Feldman. Multidimensional data visualization. *Behavior Research Methods*, 34:158–162, 2002.

[Sebrechts et al. 1999] M. M. Sebrechts, J. V. Cugini, S. J. Laskowski, J. Vasilakis, and M. S. Miller. Visualization of search results: a comparative evaluation of text, 2D, and 3D interfaces. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '99, 1999.

[Shneiderman 1996] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *IEEE Symposium on Visual Languages*, pages 336–343. IEEE, 1996.

[SIGAR] SIGAR. Hyperic SIGAR API. `http://www.hyperic.com/products/sigar`, 2012. Abgerufen 12.04.2012.

[SWIG] SWIG. Simplified Wrapper And Interface Generator. `http://www.swig.org/`, 2012. Abgerufen 18.04.2012.

[Tominski 2011] C. Tominski. Event-based concepts for user-driven visualization. *Information Visualization*, 10(1):65, 2011.

[Tominski et al. 2005] C. Tominski, J. Abello, and H. Schumann. Interactive poster: 3D axes-based visualizations for time series data. In *Poster Compendium of IEEE Symp. on Information Visualization (InfoVis' 05)*. Citeseer, 2005.

[van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM, Apr. 2012.

[Vijgen 2009] R. Vijgen. The Deleted City. `http://deletedcity.net`, 2009. Abgerufen 17.02.2012.

[Völz 1999] H. Völz. *Das Mensch-Technik-System.: Physiologische, physikalische und technische Grundlagen - Software und Hardware.* Expert-Verl., 1999.

[Waller and Hasselbring 2012] J. Waller and W. Hasselbring. A comparison of the influence of different multi-core processors on the runtime overhead for application-level monitoring. In *Multicore Software Engineering, Performance, and Tools (MSEPT)*, pages 42–53. Springer, June 2012.

[Wettel and Lanza 2007] R. Wettel and M. Lanza. Visualizing Software Systems as Cities. 2007.

[Wettel and Lanza 2008] R. Wettel and M. Lanza. Codecity: 3d visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*, pages 921–922. ACM, 2008.

[Wiki2011] Wiki2011. WASD. `http://de.wikipedia.org/wiki/WASD`, December 2011. Abgerufen 14.05.2012.

[Worley 2005] P. H. Worley. MPICL. `http://www.csm.ornl.gov/picl/`, March 2005. Abgerufen 29.05.2012.

[Xu and Robbins 1996] J. Xu and K. Robbins. *Using the Tools of Parallel Program Performance Evaluation to Visualize Scientific Data.* Citeseer, 1996.

[Yazel et al. 2012] D. Yazel et al. Xith3D. `http://xith.org/`, 2012. Abgerufen 23.04.2012.