Statistische Modellering van Computerprogramma's

Accurate Statistical Workload Modeling

Lieven Eeckhout

Promotor: Prof. dr. ir. K. De Bosschere

Proefschrift ingediend tot het behalen van de graad van
Doctor in de Toegepaste Wetenschappen:
Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: Prof. dr. ir. J. Van Campenhout
Faculteit Toegepaste Wetenschappen
Academiejaar 2002–2003

Aan "pépé Brussel"

Voorwoord

Het vervolmaken van een doctoraat is een werk van lange adem dat haast onmogelijk is als alleenstaand individu. Zonder de steun van een aantal mensen zou ik niet staat geweest zijn dit doctoraat neer te leggen. Op deze plaats wil ik hen dan ook uitvoerig bedanken.

Allereerst wil ik mijn promotor prof. Koen De Bosschere bedanken. Tijdens de lessen Computerarchitectuur van mijn opleiding burgerlijk ingenieur wist Koen mijn interesse te wekken voor het vakgebied door zijn gedreven manier van lesgeven. Het is dan ook geen toeval dat dit doctoraat zich precies in dit domein situeert. Tijdens het laatste jaar van mijn studies bood Koen mij de mogelijkheid aan om doctoraatsonderzoek aan te vatten. Dit sprak mij zo sterk aan dat ik op zijn voorstel ben ingegaan. Eerlijk gezegd, ik heb er nog geen seconde spijt over gehad. Dit uiteraard door de niet aflatende steun en motivatie van Koen. Steeds was Koen beschikbaar en bereid te discussiëren over een technisch onderwerp, een artikel in opbouw, een praktische regeling, noem maar op. Het heeft mij telkens moed gegeven dat Koen geloofde in mijn capaciteiten en mijn werk. Waarvoor nogmaals van harte bedankt.

In tweede instantie wil ik dr. Henk Neefs bedanken die mij ingeleid heeft in de knepen van het onderzoek in de computerarchitectuur. Reeds tijdens mijn scriptie wist Henk mij op een bijzondere manier te motiveren en in de beginfase van mijn doctoraat heeft hij mij op het goede spoor gezet door mij te helpen bij het schrijven van artikels. Daarnaast steunde hij mijn keuze om het gedrag van computerprogramma's te bestuderen. Het doet mij dan ook een groot plezier dat Henk van het zonnige Californië overvloog naar het grijze België om in mijn doctoraatsjury te zetelen.

Mijn bureaugenoten Bart, Hans en Veerle wil ik bedanken voor hun steun, de sfeer en de tijd die ze gespendeerd hebben om onderzoeksonderwerpen te bespreken. Dit heeft zonder enige twijfel bijgedragen tot dit werk. In het bijzonder wil ik Hans bedanken voor de talrijke boeiende discussies die uiteindelijk geleid hebben tot de resultaten beschreven in hoofdstuk 6.

Verder wil ik de overige mensen uit mijn directe werkomgeving be-

danken, meer bepaald de mensen uit de onderzoeksgroep, die via de review-bijeenkomsten feedback hebben gegeven op mijn werk. Vanzelfsprekend wil ik hen ook bedanken voor de collegialiteit die er voor gezorgd heeft dat er een aangename werksfeer heerst in onze onderzoeksgroep. Bedankt Joni, Marnik, Wim, Peter, Michiel De Wilde, Hendrik, Ronny, Michiel Ronsse, Bjorn, Mark, Bruno, Kristof, Andy, prof. Dirk Stroobandt en in het bijzonder prof. Jan Van Campenhout en prof. Erik D'Hollander om ook in mijn jury te zetelen.

I would like to thank prof. James E. Smith from the University of Wisconsin–Madison for being willing to serve as a member of my PhD committee. I'm really honoured to have Jim on my committee for two reasons. First of all, I think everybody working in this domain would be honoured having Jim in their PhD committee given his expertise in computer architecture making of him an authority on the field. Second, Jim was the first one to publish a paper on statistical simulation which is the main topic of this dissertation. When I started working on my PhD, I read the paper Jim published with a student of him, Richard Carl, in the Performance Analysis and its Impact on Design (PAID) workshop in 1998. The paper was entitled 'Modeling Superscalar Processors via Statistical Simulation' and was so interesting to me that I decided to start working in this area. This research topic has become the major part of my dissertation.

I also would like to thank dr. Pradip Bose from IBM T.J. Watson Research Center for being willing to serve on my PhD committee as well. Pradip has been active in the field of early design stage power/performance modeling techniques for many years. This research topic is exactly what this dissertation is dealing with. Pradip has always shown great interest in my work as he served in several program committees of conferences I submitted papers to. In particular, Pradip was the track co-chair and the program chair of the International Symposium on Performance Analysis of Systems and Software (ISPASS) in 2000 and 2001, respectively. In each of these two conferences, I got a paper accepted which meant a lot to me since it made me feel that one day I would be able to present my dissertation.

Daarnaast wil ik ook de overige leden van de doctoraatsjury bedanken voor de moeite die ze zich getroost hebben om mijn werk onder de loep te nemen. Bedankt prof. Bart Dhoedt en prof. Jean-Pierre Ottoy.

Sta mij ook toe het Instituut voor de aanmoediging van Innovatie door Wetenschap en Technologie in Vlaanderen, kortweg het IWT, te bedanken voor hun financiële ondersteuning in de vorm van een specialisatiebeurs.

Uiteraard wil ik ook mijn familie bedanken voor hun steun en toeverlaat. In het bijzonder dank ik hier mijn ouders, Renaat, Goedele en Bernard, die mijn werkzaamheden, ieder op hun eigen manier, altijd met veel belangstelling gevolgd hebben. Mijn ouders hebben altijd geloofd in mijn mogelijkheden en hebben mij telkens advies gegeven wanneer er belangrijke beslissingen genomen moesten worden. Eens ik een bepaalde keuze gemaakt had, werd ik daar dan ook ten volle in gesteund. Ma en pa, dit doctoraat komt voor een groot deel voort uit de bagage die ik van jullie heb meegekregen.

Ook wil ik de familie van Hannelore bedanken die ondertussen, weliswaar niet officieel maar wel in mijn hart, ook mijn familie geworden is. Bedankt Chris, Marie-Paule, Stine, Jurgen, Nina, Erika, Steven, Margo, Xander en de grootouders en -tante van Kuurne en Kruishoutem.

Ten slotte wil ik Hannelore bedanken. Het onder woorden brengen van waarvoor ik haar wil bedanken zou alleen maar afbreuk doen aan waar het echt over gaat. Maar laat mij toch toe het volgende te zeggen: dank je voor de liefde die je onze kleine Zoë en mezelf elke dag opnieuw geeft!

Mijn taak zit erop, het is nu aan de lezer om mijn stappen in dit onderzoek na te lopen. Voor degenen die niet de volle tocht willen ondernemen, wil ik de verkorte route in de vorm van de Nederlandse samenvatting aanbevelen.

Lieven Eeckhout
25 november 2002
Gent

Examencommissie

Voorzitter:	Prof. P. Kiekens, Decaan FTW, Universiteit Gent
Secretaris:	Prof. J. Debacker FTW, Universiteit Gent
Leden:	Dr. P. Bose IBM T.J. Watson Research Center, Yorktown Heights, NY
	Prof. K. De Bosschere ELIS, FTW, Universiteit Gent
	Prof. B. Dhoedt INTEC, FTW, Universiteit Gent
	Prof. E. D'Hollander ELIS, FTW, Universiteit Gent
	Dr. H. Neefs Intel, Santa Clara, CA
	Prof. J.-P. Ottoy BIOMATH, FLTBW, Universiteit Gent
	Prof. J. E. Smith ECE, University of Wisconsin–Madison, WI
	Prof. J. Van Campenhout ELIS, FTW, Universiteit Gent

Inhoudsopgave

1 Inleiding . . . 1
- 1.1 Het ontwerp van een microprocessor . . . 2
- 1.2 Bijdragen . . . 3
- 1.3 Overzicht . . . 5

2 Statistische simulatie . . . 5
- 2.1 Statistische profilering . . . 7
- 2.2 Generatie en simulatie van synthetische-instructiestromen . . . 9
- 2.3 Toepassingen . . . 12
- 2.4 Nauwkeurigheid . . . 13

3 Verhogen van de nauwkeurigheid . . . 18

4 Registerafhankelijkheden . . . 20

5 Vermogenschatting . . . 23

6 Ontwerp van een werklast . . . 26

7 Conclusie . . . 31

Bibliografie **33**

Nederlandstalige samenvatting

1 Inleiding

Hedendaagse microprocessors worden steeds complexer. De wet van Moore stelt immers dat de prestatie van de microprocessor om de 18 maanden verdubbelt. Deze evolutie wordt door een drietal belangrijke factoren mogelijk gemaakt. Ten eerste laat de chiptechnologie toe honderden miljoenen transistors op één chip te plaatsen. Ten tweede ontwikkelen computerarchitecten geavanceerde microarchitecturale technieken om deze grote hoeveelheden transistors op een zo efficiënt mogelijke manier te benutten en zo de prestatie van de microprocessor maximaal op te drijven. Computerarchitecten trachten de prestatie te verhogen door allerlei vormen van parallellisme en speculatie toe te voegen. Ten derde zijn hedendaagse compilers in staat sterk geoptimaliseerde code te genereren.

Een ander belangrijk fenomeen dat zich heden ten dage manifesteert is de toenemende complexiteit van hedendaagse applicaties. Deze trend wordt veroorzaakt door de immer toenemende prestatie van computersystemen en door de steeds hogere vereisten van de eindgebruikers.

Dit heeft uiteraard zijn repercussies op de ontwerpsmethodologieën. Tegenwoordig volstaat het niet meer een nieuw computersysteem te ontwerpen op basis van intuïtie, ervaring en vuistregels. Gedetailleerde simulaties (o.a. op architecturaal niveau) van een groot aantal applicaties zijn vereist om het specifieke gedrag van een gegeven processorconfiguratie te karakteriseren. Bovendien moet een grote ontwerpsruimte geëxploreerd worden teneinde het optimale ontwerp te identificeren. En door de toenemende complexiteit van zowel processorarchitecturen als applicaties wordt dit een steeds moeilijkere taak.

Het waarheidsgetrouw modelleren van steeds complexer wordende processorarchitecturen leidt onvermijdelijk tot tragere simulatoren; het evalueren van steeds complexer wordende applicaties vereist het simuleren van steeds meer instructies (tegenwoordig worden reeds tientallen miljarden instructies gesimuleerd per applicatie). Hieruit kunnen we besluiten dat het ontwerpen van toekomstige processorarchitecturen een moeilijke en bovendien ook tijdrovende taak is.

We hebben derhalve nood aan technieken die dit proces kunnen versnellen en die ons meer inzicht kunnen geven in intrinsieke kenmerken van programma's zodat het aantal te simuleren programma's beperkt kan worden tot een minimum. Dit vormt dan ook het onderwerp van studie van deze doctoraatsthesis.

1.1 Het ontwerp van een microprocessor

Het ontwerpsproces van een microprocessor neemt typisch enkele jaren in beslag en bestaat dan ook uit een aantal belangrijke stappen [2]: samenstellen van een werklast, exploratie van de ontwerpsruimte, definitie van de microarchitectuur op basis van hoog-niveausimulaties, modellering en simulatie op laag niveau (RTL, logisch, circuit) en tenslotte validatie van de verschillende abstractieniveaus. In deze paragraaf gaan we iets dieper in op de eerste drie stappen omdat deze het onderwerp vormen van deze thesis.

In de eerste stap wordt een werklast samengesteld. Dit betekent dat een aantal computerprogramma's gekozen worden die representatief zijn voor het toepassingsgebied van de te ontwerpen microprocessor. Bijvoorbeeld, de werklast van een microprocessor die bedoeld is voor algemeen gebruik zal bestaan uit een tekstverwerker, een rekenblad, enz. Een microprocessor voor wetenschappelijke doeleinden zal ontworpen worden op basis van een werklast bestaande uit rekenintensieve applicaties. Het is duidelijk dat het selecteren van een representatieve werklast een cruciale stap is in het ontwerpsproces vermits het volledige ontwerpsproces hierop gebaseerd zal zijn. Indien een microprocessor ontworpen zou worden op basis van een niet-representatieve werklast, kan dit immers leiden tot een suboptimaal ontwerp.

De tweede stap bestaat uit het exploreren en afbakenen van de ontwerpsruimte. Dit gebeurt in samenspraak met experten uit andere domeinen, b.v. met experten op het gebied van chiptechnologie. Merk op dat dit uiterst belangrijke aspecten zijn binnen de context van hedendaagse en toekomstige chiptechnologieën [27] waarbij vertragingstij-

den tengevolge van bedrading en vermogenverbruik een steeds groter wordende impact hebben op het ontwerp en de totale prestatie van het systeem.

In de derde stap wordt de microarchitectuur gedefinieerd. Dit gebeurt op basis van uitvoerige simulaties op architecturaal niveau[1]. Dit simulatieproces duurt ontzettend lang omdat een grote ontwerpsruimte geëvalueerd moet worden. Het aantal architecturale parameters dat gevarieerd kan worden loopt snel in de tientallen waardoor een zeer grote ontwerpsruimte ontstaat. Daarnaast moet men ook rekening houden met tal van technologische aspecten. Bovendien moet ook een voldoend grote werklast gesimuleerd worden waarbij iedere benchmark bestaat uit een zeer groot aantal instructies, b.v. een paar miljard instructies per benchmark.

We kunnen dus besluiten dat een aantal belangrijke aspecten verbonden zijn aan de eerste stappen van het ontwerpsproces van een microprocessor:

- het selecteren van representatieve benchmarks met bijhorende input
- snelle exploratie van de ontwerpsruimte
- rekening houden met technologische aspecten zoals vermogenverbruik en vertragingstijden ten gevolge van bedrading

en dit alles zonder daarbij aan al te veel nauwkeurigheid in te boeten. M.a.w. we willen correcte ontwerpsbeslissingen nemen in een vroeg stadium van het ontwerp. Dit zal de ontwerpskost en de ontwerpstijd aanzienlijk verkorten.

1.2 Bijdragen

Deze doctoraatsthesis maakt de volgende bijdragen:

- We evalueren statistische simulatie als mogelijk hulpmiddel bij de exploratie van de ontwerpsruimte in een vroeg stadium van het ontwerpsproces. De basisidee van statistische simulatie is eenvoudig: we meten een aantal programmakarakteristieken op van de uitvoering van een computerprogramma, we genereren vervolgens een synthetische-instructiestroom gebruik makend van

[1] Een wijdverspreid voorbeeld van een architecturale simulator is de SimpleScalar Tool Set [1].

deze karakteristieken en simuleren ten slotte deze synthetische-instructiestroom. De prestatiemaat die berekend wordt op deze manier is een schatting voor de prestatie van het oorspronkelijk programma. We evalueren zowel de absolute als de relatieve nauwkeurigheid. Bovendien tonen we ook aan dat statistische simulatie een snelle simulatietechniek is.

- We verhogen de nauwkeurigheid van statistische simulatie door een aantal verbeteringen voor te stellen [10, 16, 17]: (i) het gebruik van hogere-orde distributies voor het karakteriseren van afhankelijkheden tussen instructies, (ii) de implementatie van een terugkoppellus in het algoritme voor het genereren van synthetische-instructiestromen zodat de syntactische correctheid gegarandeerd kan worden, en (iii) het modelleren van geclusterde cache misses.

- We tonen aan dat de distributies met betrekking tot de registerafhankelijkheden tussen instructies over het algemeen aan een machtswet voldoen [15]. Deze informatie wordt vervolgens aangewend om een hybride analytisch-statistische methodologie voor te stellen die bijna even nauwkeurig is als statistische simulatie. Bovendien demonstreren we dat deze methode ons toelaat exploraties te doen in de ruimte van de computerprogramma's, m.a.w. de verschillende parameters die een programma karakteriseren kunnen vrij gevarieerd worden.

- We tonen aan dat statistische simulatie ook aangewend kan worden voor het schatten van het vermogenverbruik in een vroeg stadium van het ontwerpsproces [14]. Bovendien blijkt deze methode ook bruikbaar te zijn om de invloed op het vermogenverbruik na te gaan van de verschillende programmakarakteristieken.

- Ten slotte stellen we ook een techniek voor voor het selecteren van representatieve computerprogramma's met bijhorende inputs [23, 24, 25, 26]. Deze techniek is gebaseerd op multivariate statistische technieken.

Naast de onderwerpen die besproken zullen worden in deze thesis werd nog ander onderzoek verricht gedurende de voorbije vier jaar. Ten eerste hebben we statistische simulatie gebruikt bij de evaluatie van een experimentele microarchitectuur [9, 12, 18, 19, 20, 21, 22], namelijk de blokgestructureerde microarchitectuur [32, 33]. Ten tweede werden

een aantal onderwerpen behandeld binnen het domein van de karakterisatie van computerprogramma's: analyse van het niet-uniform gedrag van programma's [13] en het kwantificeren van verschillen in het gedrag van multimediale programma's versus programma's van algemeen gebruik [11, 39].

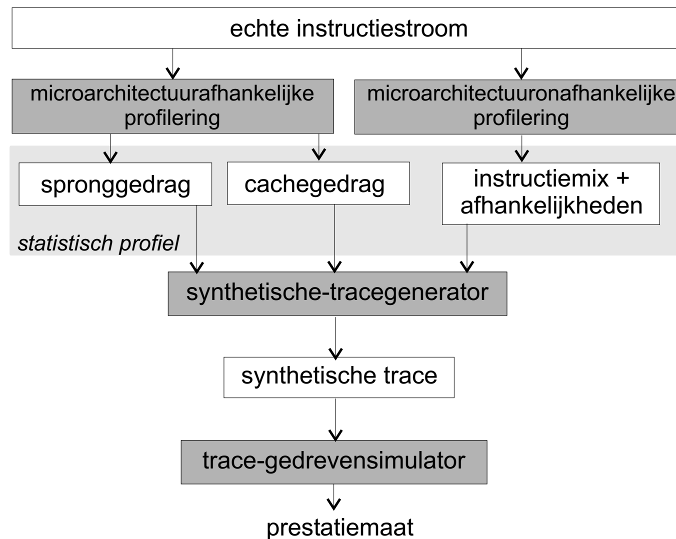
1.3 Overzicht

Deze nederlandstalige samenvatting is als volgt opgebouwd, volledig gelijklopend aan de eigenlijke doctoraatsthesis. Paragraaf 2 bespreekt statistische simulatie: algemeen raamwerk, toepassingen en evaluatie a.d.h.v. absolute en relatieve nauwkeurigheden. In paragraaf 3 behandelen we twee technieken om de nauwkeurigheid van statistische simulatie te verhogen: het modelleren van de afhankelijkheden tussen instructies m.b.v. hogere-orde distributies, en het modelleren van clusters van cache misses. In paragraaf 4 tonen we aan dat de distributie van de registerafhankelijkheden aan een machtswet voldoet en geven we aan hoe die notie aangewend kan worden in een analytisch model. Dat statistische simulatie een nauwkeurige techniek is voor het modelleren van het vermogenverbruik van een microarchitectuur wordt behandeld in paragraaf 5. Paragraaf 6 licht de techniek toe die in staat is representatieve benchmark-input koppels te identificeren. We besluiten ten slotte in paragraaf 7.

2 Statistische simulatie

Het algemeen raamwerk van statistische simulatie wordt weergegeven in figuur 1.1 en bestaat uit drie stappen: *statistische profilering*, *genereren van een synthetische-instructiestroom* en *simulatie van deze synthetische-instructiestroom*.

In een eerste stap wordt de instructiestroom (*trace*) van een bestaande applicatie geanalyseerd. Dit gebeurt m.b.v. twee hulpprogramma's waarvan er één architectuurafhankelijke programmakarakteristieken opmeet en waarvan er één architectuuronafhankelijke karakteristieken opmeet. Het hulpprogramma dat de architectuuronafhankelijke karakteristieken opmeet, meet de verdeling van de instructietypes op alsook de distributies betreffende de afhankelijkheden tussen de individuele instructies. Deze distributies worden vervolgens opgeslagen in een intermediair bestand. De hulpprogramma's die de architectuurafhankelijke programmakarakteristieken opmeten, berekenen distributies be-



Figuur 1.1: Statistische simulatie: raamwerk.

treffende het spronggedrag en het cachegedrag van de applicatie. Deze karakteristieken zijn specifiek voor de gekozen applicatie, de gekozen sprongvoorspeller en de gekozen cacheconfiguratie. De verzameling van architectuurafhankelijke en architectuuronafhankelijke programmakarakteristieken wordt een *statistisch profiel* genoemd en modelleert in feite de uitvoering van een applicatie op een statistische wijze.

Merk op dat deze karakteristieken berekend kunnen worden op basis van een instructiestroom die opgeslagen is op een harde schijf. Maar het is allicht meer aangewezen deze karakteristieken *on-the-fly* te berekenen m.b.v. een gemodificeerde functionele simulator of door een geïnstrumenteerde versie van de applicatie uit te voeren op een echt systeem. Een tweede belangrijke opmerking is dat het berekenen van een statistisch profiel, wat het overlopen van de volledige instructiestroom vereist, slechts één keer dient te gebeuren. Rekening houdend met het feit dat het simuleren van een synthetische-instructiestroom grootteordes sneller is dan het simuleren van de echte instructiestroom, is het opmeten van een statistisch profiel de moeite waard.

Eenmaal een statistisch profiel berekend is, wordt een synthetische-instructiestroom gegenereerd a.d.h.v. dit statistisch profiel. De synthetische-instructiestroom heeft dezelfde eigenschappen als de oorspronkelijke instructiestroom waarvan het statistisch profiel gegenereerd werd.

Dit geldt uiteraard enkel voor de programmakarakteristieken die opgenomen zijn in het statistisch profiel. De synthetische-instructiestroom kan vervolgens gesimuleerd worden door een simulator. Indien het statistisch profiel de relevante programmakarakteristieken bevat, zullen de uitvoeringskarakteristieken van de oorspronkelijke applicatie en de synthetische-instructiestroom vergelijkbaar zijn.

Idealiter zouden we willen dat alle programmakarakteristieken in een statistisch profiel architectuuronafhankelijk zijn opdat de volledige ontwerpsruimte geëxploreerd zou kunnen worden a.d.h.v. één enkel statistisch profiel. Daarbij zouden we verschillende microarchitecturale parameters kunnen variëren zoals het aantal functionele eenheden, de grootte van het instructievenster, uitvoeringslatentie van instructies, de pijplijndiepte, de grootte van de verschillende caches, de sprongvoorspeller, enz. Jammer genoeg zijn programma-eigenschappen met betrekking tot lokaliteit, zoals cache- en spronggedrag, moeilijk te modelleren m.b.v. architectuuronafhankelijke karakteristieken. Daarom stellen we hier een pragmatische oplossing voor door een onderscheid te maken tussen architectuurafhankelijke en -onafhankelijke karakteristieken. Deze opsplitsing laat ons desalniettemin toe een groot deel van de ontwerpsruimte te exploreren a.d.h.v. één enkel statistisch profiel.

2.1 Statistische profilering

Bij het zoeken naar een statistisch profiel moeten drie belangrijke aspecten in beschouwing genomen worden. Ten eerste moet het statistisch profiel de meest relevante distributies bevatten opdat het beschikbare parallellisme in programma's op een waarheidsgetrouwe manier gemodelleerd zou worden in de synthetische-instructiestroom. Ten tweede mag het statistisch profiel niet te ingewikkeld zijn teneinde de opslag van een statistisch profiel te beperken. M.a.w. het aantal distributies dat opgenomen wordt in een statistisch profiel moet beperkt blijven zonder daarbij aan nauwkeurigheid in te boeten. Ten derde wensen we synthetische-instructiestromen te genereren die syntactisch correct zijn opdat deze instructiestromen uitgevoerd zouden kunnen worden op bestaande instructiestroomgedreven simulatiesoftware. Bovendien zal een syntactisch correcte synthetische-instructiestroom het gedrag van een echte instructiestroom beter weerspiegelen. Met syntactisch correct zijn wordt bedoeld dat een schrijfinstructie (*store*) en een sponginstructie geen doeloperand mag hebben en dat een operatie geen vier inputoperandi kan hebben. Deze voorwaarde wordt niet gegarandeerd

in aanverwant werk [6, 34, 35, 37].

In de volgende twee paragrafen worden respectievelijk de microarchitectuuronafhankelijke en de -afhankelijke karakteristieken in beknopte vorm aangegeven.

Microarchitectuuronafhankelijke karakteristieken

Deze programmakarakteristieken zijn onafhankelijk van de microarchitectuur en beschrijven in feite de hoeveelheid parallellisme die aanwezig is in een uitvoeringsstroom zoals die aan een processor aangeboden wordt. Deze karakteristieken zijn echter wel afhankelijk van de applicatie, de compiler en de instructieset-architectuur (ISA).

Een eerste programmakarakteristiek is de *instructiemix* of de verdeling van de verschillende instructietypes. In ons model voor de Alpha ISA hebben we 13 instructietypes beschouwd afhankelijk van de semantiek en de uitvoeringslatentie: operaties op gehele getallen (optellingen, aftrekkingen, shiftoperaties, logische operaties, ...), leesinstructies (*load*), schrijfinstructies (*store*), conditionele sprongen, niet-conditionele sprongen, indirecte sprongen, functieoproepen, indirecte functieoproepen, functieterugkeer, integer vermenigvuldigingen, vlottendekomma operaties, delingen in enkele precisie en delingen in dubbele precisie.

Een tweede karakteristiek is het *aantal operandi per instructietype*. Het variabel aantal operandi wordt veroorzaakt door het feit dat sommige instructies zowel in register-register als in register-constante formaat voorkomen ofschoon beide tot hetzelfde instructietype behoren.

Een derde karakteristiek die we opgemeten hebben is de distributie van de *registerafhankelijkheden*: we hebben de probabiliteit opgemeten dat een bronoperand van instructie x geproduceerd werd door instructie $x - \delta$ die δ instructies vóór instructie x komt in de instructiestroom. Het betreft hier echte data-afhankelijkheden. Valse afhankelijkheden (output- en anti-afhankelijkheden) worden niet in beschouwing genomen omdat we in de evaluatie zogenaamde *out-of-order* architecturen veronderstellen. Dit type superscalaire architectuur is de meest voorkomende architectuur voor processors voor algemeen gebruik waarbij registerhernoeming valse afhankelijkheden dynamisch verwijdert tijdens de uitvoering van een computerprogramma.

Een vierde karakteristiek meet de *geheugenafhankelijkheden* op tussen instructies. Deze karakteristiek meet de echte afhankelijkheden op tussen instructies via waarden in het geheugen, b.v. een schrijfinstructie

schrijft naar een adres in het geheugen dat vervolgens gelezen wordt door een leesinstructie.

Merk op dat sommige van de hier opgesomde distributies oneindig zijn. In de praktijk zullen we de distributies beperken tot een bepaalde maximumwaarde, namelijk N_{max}. Deze maximumwaarde stelt een limiet op de grootte van de instructievensters die gemodelleerd kunnen worden. In deze thesis stellen we $N_{max} = 512$ waarmee we architecturen van de nabije toekomst kunnen evalueren.

Microarchitectuurafhankelijke karakteristieken

De microarchitectuurafhankelijke karakteristieken hebben betrekking op het cache- en sprongvoorspellingsgedrag van een programma, zie figuur 1.1. Deze karakteristieken kunnen opgemeten worden door de instructiestromen te simuleren op eenvoudige simulatoren die het gewenste gedrag simuleren.

Een eerste karakteristiek heeft betrekking op de sprongvoorspellingsnauwkeurigheid en bestaat uit zeven probabiliteiten. Hierbij wordt een onderscheid gemaakt tussen types sprongen (b.v. conditionele sprongen, functie-oproepen, etc.) alsook tussen twee types foutieve voorspellingen (doeladres foutief voorspeld en sprongrichting foutief voorspeld).

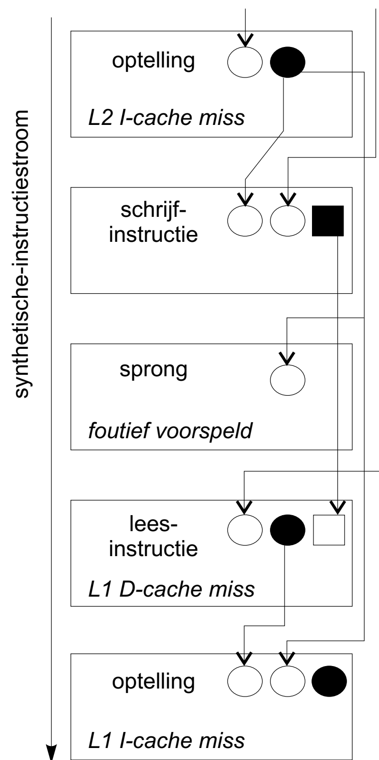
Een tweede karakteristiek betreft het gedrag van de data- en de instructiecache. In beide gevallen worden twee probabiliteiten opgenomen, namelijk de probabiliteit van een cache miss op het eerste cacheniveau, een zogenaamde L1 cache miss, en de probabiliteit van een cache miss op het tweede cacheniveau, een L2 cache miss.

2.2 Generatie en simulatie van synthetische-instructiestromen

Het genereren van een synthetische-instructiestroom gebeurt à la Monte Carlo: een randomgetal wordt gegenereerd dat vervolgens gebruikt wordt om een specifieke programmakarakteristiek te specificeren gebruik makend van de cumulatieve distributiefunctie. Het generatieproces gebeurt instructie per instructie, zie figuur 1.2:

1. bepaal het instructietype en het aantal operandi;

2. voor ieder operand, bepaal door welke instructie dit operand geproduceerd werd;



Figuur 1.2: Generatie van een synthetische-instructiestroom.

3. indien het een leesinstructie betreft, bepaal of deze instructie afhankelijk is van een voorafgaande schrijfinstructie via het geheugen;

4. indien het een spronginstructie betreft, bepaal of deze instructie al dan niet correct voorspeld zal worden door de sprongvoorspeller;

5. indien het een schrijfinstructie betreft, bepaal of de data van de L1-cache, de L2-cache of het geheugen opgehaald moeten worden;

6. bepaal of de instructie een cache miss veroorzaakt in de instructiecache.

De laatste fase van statistische simulatie is het simuleren van de synthetische-instructiestroom. Deze simulatie geeft ons een schatting van de prestatiekarakteristieken van de oorspronkelijke instructiestroom. Een belangrijke prestatiemaat is het aantal uitgevoerde instructies per klokcyclus wat eenvoudig berekend kan worden door het aantal gesimuleerde instructies te delen door het aantal gesimuleerde klokcycli. Deze prestatiemaat wordt in de vakliteratuur IPC genoemd, of *instructies per cyclus*.

Merk op dat er in stap 2 niet gegarandeerd kan worden dat het operand geproduceerd zal worden door een schrijfinstructie of een spronginstructie. Om dit op te lossen stellen we voor deze stap een aantal keer te proberen totdat een afhankelijkheid gevonden wordt die niet door een schrijfinstructie of een spronginstructie geproduceerd wordt. Indien dit niet mogelijk blijkt te zijn na 10000 keer geprobeerd te hebben, wordt de afhankelijkheid gewoon weggelaten. Uit analyse van de op deze manier gegenereerde synthetische-instructiestromen is gebleken [17] dat de (marginale) distributie van de leeftijd van de registeroperandi afwijkt van de distributie van de instructiestroom van het programma dat we wensen te modelleren. Om deze afwijking te corrigeren stellen we voor een *terugkoppellus* te introduceren in het generatiemechanisme [17]: indien de tot nu toe gegenereerde distributie afwijkt van de gewenste distributie worden waarden gegenereerd zodanig dat deze afwijking gecompenseerd wordt. Om dit mogelijk te maken genereren we waarden volgens een *foutdistributie*, d.i. de distributie die ontstaat door de tot nu gegenereerde distributie af te trekken van de gewenste distributie en opnieuw te normaliseren. Uit experimenten is gebleken dat het gestelde probleem opgelost wordt m.b.v. de

terugkoppellus. Bovendien leidt dit tot een iets hogere voorspellingsnauwkeurigheid.

2.3 Toepassingen

Statistische simulatie heeft een aantal belangrijke toepassingen:

- **Prestatie-evaluatie van een uniprocessor.** De meest voor de hand liggende toepassing van statistische simulatie is uiteraard het efficiënt schatten van de prestatie van een uniprocessor. Vermits statistische simulatie een snelle simulatietechniek is die bovendien vrij nauwkeurig is, kan deze techniek dus aangewend worden in een vroeg ontwerpsstadium. Het is geenszins de bedoeling simulaties gebruikmakend van echte instructiestromen te vervangen door statistische simulatie. Statistische simulatie is veeleer bedoeld als hulpmiddel bij het exploreren van de ontwerpsruimte, bijvoorbeeld om een kleinere regio te lokaliseren met interessante karakteristieken die dan verder geëvalueerd kan worden via gedetailleerde simulaties. Deze toepassing wordt verder besproken in paragraaf 2.4.

- **Systeemevaluatie.** Bij grotere systemen bestaande uit meerdere processors, zoals bijvoorbeeld multiprocessors, clusters van computers, e.d., is simulatietijd nog een veel groter probleem dan reeds het geval is voor uniprocessorsystemen. In [36] geven Nussbaum en Smith een voorbeeld hoe statistische simulatie gebruikt kan worden bij de evaluatie van symmetrische-multiprocessorsystemen.

- **Karakterisatie van computerprogramma's.** Tijdens het validatieproces van statistische simulatie zal duidelijk worden welke karakteristieken belangrijk zijn bij het modelleren van computerprogramma's. M.a.w. er zal een differentiatie gemaakt worden tussen karakteristieken die een invloed hebben op de prestatie en karakteristieken die geen invloed hebben. De karakteristieken die een invloed hebben op de prestatie moeten bijgevolg opgenomen worden in het statistisch profiel. Dit wordt geïllustreerd in paragrafen 3 en 4.

- **Exploratie van de ruimte van computerprogramma's.** De programmakarakteristieken die opgenomen worden in een statistisch profiel kunnen vrij gevarieerd worden. Op deze manier kan de

invloed nagegaan worden van de verschillende programmakarakteristieken op de prestatie. In paragraaf 4 wordt hiervan een mooi voorbeeld gegeven.

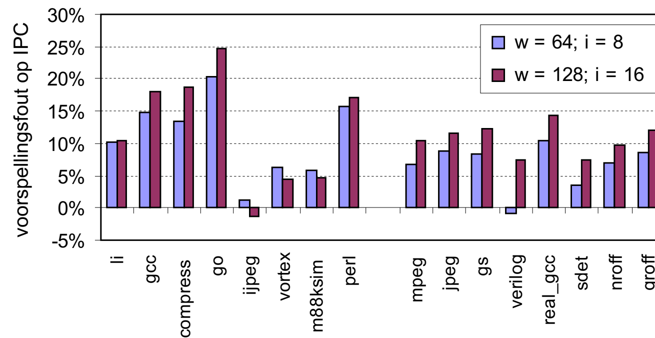
- **Schatten van vermogenverbruik.** Vermogenverbruik is een belangrijke ontwerpsparameter bij het ontwerp van hedendaagse microprocessors[2] door de steeds toenemende integratie en klokfrequenties van hedendaagse en toekomstige chiptechnologieën. Daarom stellen we voor in paragraaf 5 om statistische simulatie te combineren met vermogenschatting op architecturaal niveau. Op die manier kan vermogenverbruik in rekening gebracht worden in vroege stadia van het ontwerpsproces.

2.4 Nauwkeurigheid

In deze paragraaf bespreken we de nauwkeurigheid die haalbaar is via statistische simulatie bij het schatten van de prestatie van een microprocessor. Daartoe hebben we twee benchmark suites gebruikt, namelijk SPECint95[3] en IBS [38], beide bestaande uit 8 computerprogramma's. De SPECint95 instructiestromen werden gegenereerd op een Alpha systeem; de IBS instructiestromen werden gegenereerd op een MIPS systeem. De architectuur die we verondersteld hebben bij de evaluatie is een *out-of-order* architectuur wat betekent dat instructies niet noodzakelijk in programmavolgorde uitgevoerd zullen worden. Twee belangrijke parameters die regelmatig terug zullen komen in de hierna volgende bespreking, zijn: w de grootte van het instructievenster, kortweg de *venstergrootte*, of het maximaal aantal instructies dat op een gegeven moment in verwerking kan zijn, en i het aantal instructies dat per klokcyclus geselecteerd kunnen worden om uitgevoerd te worden, of nog de *uitvoeringsbandbreedte*. Een processorconfiguratie zal als volgt samengevat worden: w/i. Een dergelijk type superscalaire architectuur is geïmplementeerd in bijna alle hedendaagse microprocessors voor algemeen gebruik, b.v. de Alpha 21264 [31], de MIPS R10000 [42], de Pentium 4 [29], enz. Voor een meer gedetailleerde bespreking van de methodologie—betreffende de benchmarks, de cache-configuraties, de sprongvoorspeller, de uitvoeringslatenties, enz.–die gebruikt werd bij de evaluatie van statistische simulatie verwijzen we naar de eigenlijke doctoraatsthesis.

[2]Dit is niet enkel het geval voor ingebedde processors; dit is ook een reëel probleem voor zogenaamde *high-performance* microprocessors [28].

[3]`http://www.spec.org`



Figuur 1.3: Absolute voorspellingsnauwkeurigheid.

De nauwkeurigheid wordt gemeten a.d.h.v. de *voorspellingsfout op de IPC*:

$$\textit{voorspellingsfout op de IPC} = \frac{\textit{IPC synthetische trace} - \textit{IPC echte trace}}{\textit{IPC echte trace}}. \tag{1.1}$$

M.a.w. de voorspellingsfout op de IPC meet de procentuele afwijking van de geschatte IPC (opgemeten via statistische simulatie) tov. de IPC van de echte instructiestroom.

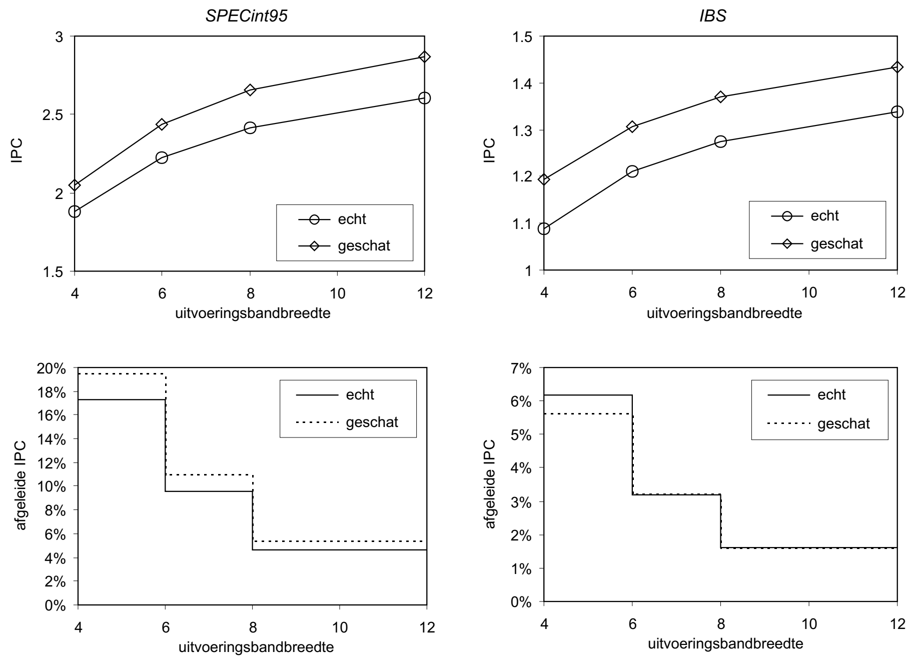
In figuur 1.3 wordt de voorspellingsfout op de IPC weergegeven voor de verschillende benchmarks en dit voor twee verschillende processorconfiguraties, namelijk 64/8 en 128/16. De voorspellingsfout op de IPC is over het algemeen niet hoger dan 15% tot 20%. Bijvoorbeeld, voor de IBS traces (de rechterhelft van de grafiek) is de gemiddelde voorspellingsfout 6,8% voor de 64/8 processorconfiguratie en 10,7% voor de 128/16 processorconfiguratie; voor de SPECint95 traces (de linkerhelft van de grafiek) is dit respectievelijk 11% en 12,4%.

In de eigenlijke doctoraatsthesis wordt de invloed van de verschillende programmakarakteristieken (instructiemix, afhankelijkheden tussen instructies, spronggedrag, cachegedrag) op de nauwkeurigheid uitvoerig besproken. Uit deze analyse blijkt dat de voorspellingsfout vooral te wijten is aan het modelleren van de afhankelijkheden tussen instructies en aan het modelleren van het cachegedrag. In de volgende paragraaf zullen technieken gesproken worden die deze foutbronnen trachten te reduceren. Het statistisch modelleren van het spronggedrag daarentegen blijkt heel nauwkeurig te gebeuren. Naast deze individuele foutbronnen wordt ook een deel van de voorspellingsfout veroorzaakt

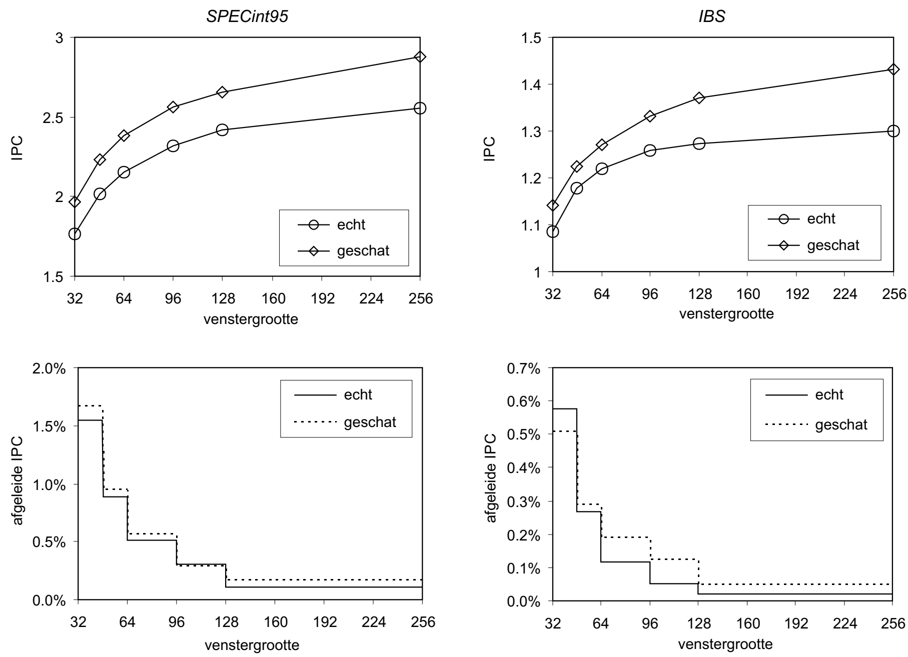
door de interactie tussen de verschillende programmakarakteristieken. Bijvoorbeeld in figuur 1.3 merken we een voorspellingsfout op voor de benchmark li van ongeveer 10%. Deze voorspellingsfout is volgens onze analyse te wijten aan de interactie tussen de afhankelijkheden tussen instructies en het datacachegedrag.

Het evaluatiecriterium dat tot nu toe gehanteerd werd, namelijk de voorspellingsfout, meet in feite de absolute nauwkeurigheid van een voorspellingstechniek. M.a.w. dit criterium is een maat voor de afwijking tussen de geschatte waarde en de werkelijke waarde in één enkel punt in de ontwerpsruimte. Ofschoon absolute nauwkeurigheid een belangrijk evaluatiecriterium is voor voorspellingstechnieken, is relatieve nauwkeurigheid binnen de context van het architecturaal ontwerp minstens even belangrijk zoniet belangrijker. Met relatieve nauwkeurigheid wordt bedoeld dat de voorspellingstechniek in staat zou moeten zijn een (prestatie)trend tussen verschillende punten in de ontwerpsruimte nauwkeurig te voorspellen. Indien een hoge relatieve nauwkeurigheid bereikt kan worden, is deze techniek bijzonder nuttig om bepaalde afwegingen te maken tijdens het architecturaal ontwerp. Bijvoorbeeld, indien de toename in prestatie ten gevolge van het verhogen van een architecturale parameter niet opweegt tegen de extra kost die daarmee gepaard gaat, zal de computerarchitect de beslissing nemen deze architecturale parameter niet te verhogen. Een dergelijke beslissing kan perfect genomen worden op basis van een nauwkeurige relatieve schatting van de prestatie.

Figuren 1.4 en 1.5 evalueren de relatieve nauwkeurigheid als functie van respectievelijk het maximum aantal te selecteren instructies per klokcyclus en de grootte van het instructievenster. In de doctoraatsthesis wordt ook de relatieve nauwkeurigheid geëvalueerd als functie van de pijplijndiepte (interessant in combinatie met een grondplanontwerp van de processorchip), het aantal op te halen instructies per klokcyclus en de latentie van schrijfinstructies (of m.a.w. de toegangstijd tot de L1 datacache). In beide figuren wordt zowel een grafiek afgebeeld die de IPC weergeeft als functie van de architecturale parameter in kwestie, als een grafiek die de afgeleide van de IPC voorstelt. De afgeleide IPC is uiteraard een getrapte functie vermits de IPC curve een stukgewijs lineaire curve is. Deze figuren tonen aan dat ofschoon er een absolute afwijking bestaat tussen de ‘echte’ curve en de ‘geschatte’ curve, de trend tussen beide behoorlijk nauwkeurig geschat wordt. Merk ook op dat ofschoon, in het geval van de IBS traces als functie van de grootte van het instructievenster, de ‘geschatte’ helling dubbel zo groot is als



Figuur 1.4: Relatieve nauwkeurigheid: IPC en de afgeleide IPC opgemeten via simulatie van de echte instructiestroom ('echt') versus de synthetische-instructiestroom ('geschat') als functie van het aantal instructies dat geselecteerd kan worden per klokcyclus.



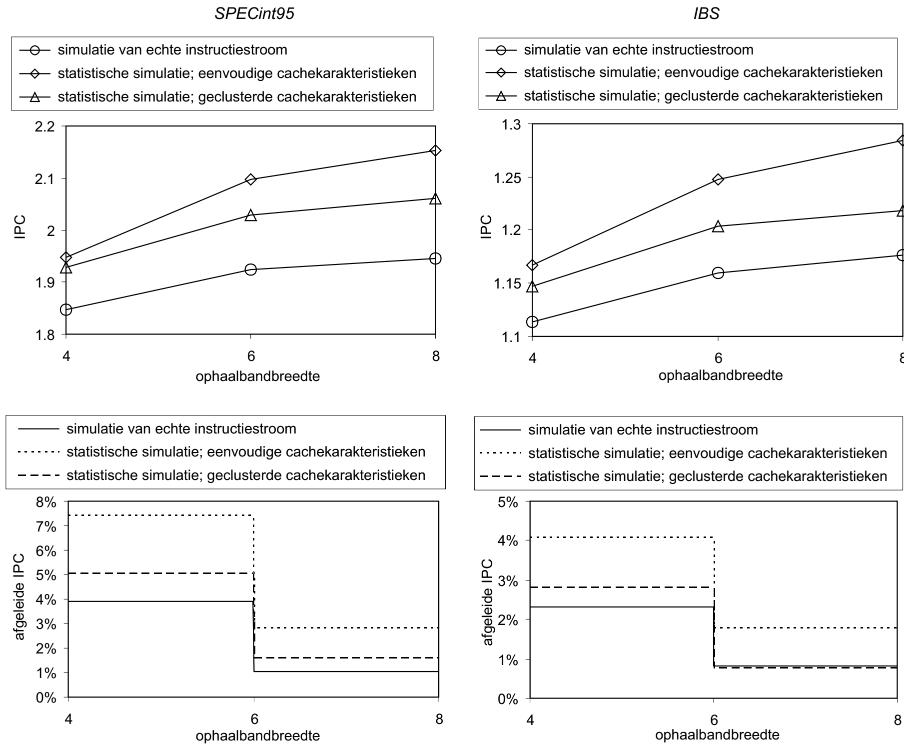
Figuur 1.5: Relatieve nauwkeurigheid: IPC en de afgeleide IPC opgemeten via simulatie van de echte instructiestroom ('echt') versus de synthetische-instructiestroom ('geschat') als functie van de grootte van het instructievenster.

de 'echte' helling, het hier gaat over zeer kleine hellingen (kleiner dan 0,2%). Bijgevolg zal dit geen invloed hebben de eventuele ontwerpsbeslissingen die op basis van deze 'geschatte' gegevens genomen zou kunnen worden.

Tot slot moeten we nog iets vermelden over de snelheid waarmee een statistische simulatie gebeurt. Om dit na te gaan hebben we het volgende experiment opgezet. We hebben verschillende synthetische-instructiestromen gegenereerd uitgaande van eenzelfde statistisch profiel waarbij de synthetische-instructiestroomgenerator telkens met een ander getal geïnitialiseerd wordt. Vervolgens hebben we de *jitter* gedefinieerd die als volgt berekend wordt: $2 \cdot s_i/\bar{x}_i$, met $\bar{x}_i$ en s_i de gemiddelde IPC respectievelijk de standaardafwijking na simulatie van i instructies. De onderliggende gedachte bij deze definitie is dat indien we een normale distributie veronderstellen, de IPC van een willekeurige synthetische-instructiestroom met een kans van ongeveer 95% in het interval $[\bar{x}_i - 2 \cdot s_i, \bar{x}_i + 2 \cdot s_i]$ ligt. De jitter na simulatie van 5 miljoen instructies is kleiner dan 0,75%. Bijgevolg kunnen we besluiten dat statistische simulatie een snelle convergentie bereikt en dus bruikbaar is in een vroeg ontwerpsstadium bij het verrichten van exploraties in de ontwerpsruimte—vergelijk 5 miljoen synthetische instructies t.o.v. enkele honderden miljoenen of zelfs miljarden instructies uit een echte instructiestroom.

3 Verhogen van de nauwkeurigheid

In hoofdstuk 3 van de doctoraatsthesis evalueren we twee mogelijke manieren om de nauwkeurigheid van statistische simulatie te verbeteren. In de eerste manier trachten we de modellering van de instructiemix en de afhankelijkheden tussen instructies te verfijnen door gebruik te maken van hogere-ordedistributies [17]. Daarbij gaan we uit van distributies die conditioneel zijn t.o.v. de instructietypes van de voorgaande instructies in de instructiestroom. Hierbij beperken we ons tot drie instructies om het totaal aantal op te meten distributies niet te laten exploderen. Via deze hogere-ordedistributies trachten we het geclusterd optreden van instructietypes in een instructiestroom te weerspiegelen, b.v. schrijfinstructies komen vaak voor in een cluster aan het begin van een procedure om argumenten op te slaan op de stapel. Deze hogere-ordedistributies blijken jammer genoeg niet steeds te leiden tot hogere voorspellingsnauwkeurigheid. Voor een 32/4 processorconfiguratie daalt de gemiddelde voorspellingsfout wel van 6,1% tot 4%;



Figuur 1.6: Relatieve nauwkeurigheid: IPC en afgeleide IPC i.f.v. het aantal instructies dat opgehaald wordt per klokcyclus; modellering van clusters van cache misses versus het gebruik van eenvoudige cachekarakteristieken.

voor een 128/16 configuratie daarentegen blijft de voorspellingsfout nagenoeg even groot (rond 9,5%).

Een tweede methode die besproken wordt in hoofdstuk 3 is het modelleren van clusters van cache misses [16]. Tot nu toe werd het cachegedrag van een applicatie gekarakteriseerd a.d.h.v. vier probabiliteiten, namelijk de L1 en L2 instructiecache miss rate en de L1 en L2 datacache miss rate. Uit experimenten is gebleken dat het gebruik van deze eenvoudige programmakarakteristieken leidt tot een aanzienlijk deel van de totale voorspellingsfout. Eén van de redenen hiervoor is het feit dat cache misses in echte instructiestromen typisch optreden in clusters. Indien we daarentegen gebruik maken van de hierboven vermelde eenvoudige karakteristieken, genereren we een cache-missgedrag dat helemaal geen clusters vertoont. Om dit te verhelpen stellen we dan ook

voor clusters van cache misses te modelleren. Dit kan verwezenlijkt worden door een distributie op te meten van het aantal instructies tussen twee opeenvolgende misses, of de zogenaamde *intermiss gap*. Uit experimenten is gebleken dat hierdoor de modellering van het instructiecachegedrag aanzienlijk verbeterd is, b.v. voor de IBS traces en een 128/16 processorconfiguratie daalt de voorspellingsfout van 10,7% tot 5,1%; voor de SPECint95 traces daalt de voorspellingsfout van 12,4% tot 10,7%. Om de modellering van het datacachegedrag nog verder te verbeteren stellen we ook voor *delayed hits* te modelleren. Dit blijkt voor een aantal benchmarks een aanzienlijk betere modellering op te leveren. Voor een gedetailleerde bespreking hierover verwijzen we naar de eigenlijke doctoraatsthesis.

In figuur 1.6 wordt opnieuw de relatieve nauwkeurigheid geëvalueerd van statistische simulatie: het gebruik van eenvoudige cachekarakteristieken versus het modelleren van clusters van cache misses. Deze grafiek geeft de prestatie weer als functie van het aantal instructies dat per klokcyclus opgehaald wordt. Uit deze grafiek blijkt duidelijk dat het modelleren van clusters van cache misses leidt tot een verhoogde relatieve nauwkeurigheid. Voor een verklaring van dit fenomeen verwijzen we naar de eigenlijk doctoraatsthesis.

4 Registerafhankelijkheden

In hoofdstuk 4 van de doctoraatsthesis geven we aan dat de distributie die de afhankelijkheden opmeet tussen instructies aan een machtswet voldoet of een Pareto verloop kent [15]. M.a.w. de dichtheidsfunctie is van de vorm $P\left[X=x\right]=\alpha x^{-\beta}$ met $1>\alpha>0$ de intersectie van de dichtheidsfunctie met de Y-as en $\beta>0$ de helling van de dichtheidsfunctie (een rechte) in een log-log grafiek. Om een zo goed mogelijke schatting te bereiken van de opgemeten distributies zijn we niet uitgegaan van de dichtheidsfunctie $P\left[X=x\right]$ maar van de conditionele-onafhankelijkheidsdistributie p_x [8] die als volgt gerelateerd is aan de dichtheidsfunctie:

$$P\left[X=x\right]=(1-p_x)\cdot\prod_{i=1}^{x-1}p_i,\ x\geq 1. \tag{1.2}$$

De betekenis van de conditionele-onafhankelijkheidsdistributie is als volgt: p_x is de kans dat een instructie onafhankelijk is van een instructie die x instructies ervoor optreedt in een instructiestroom gegeven het

feit dat deze instructie onafhankelijk is van de $x - 1$ tussenliggende instructies. Door Kamin, Adams en Dubey [30] werd deze distributie benaderd door een exponentiële functie:

$$p_x \approx 1 - \alpha e^{-\beta x}. \quad (1.3)$$

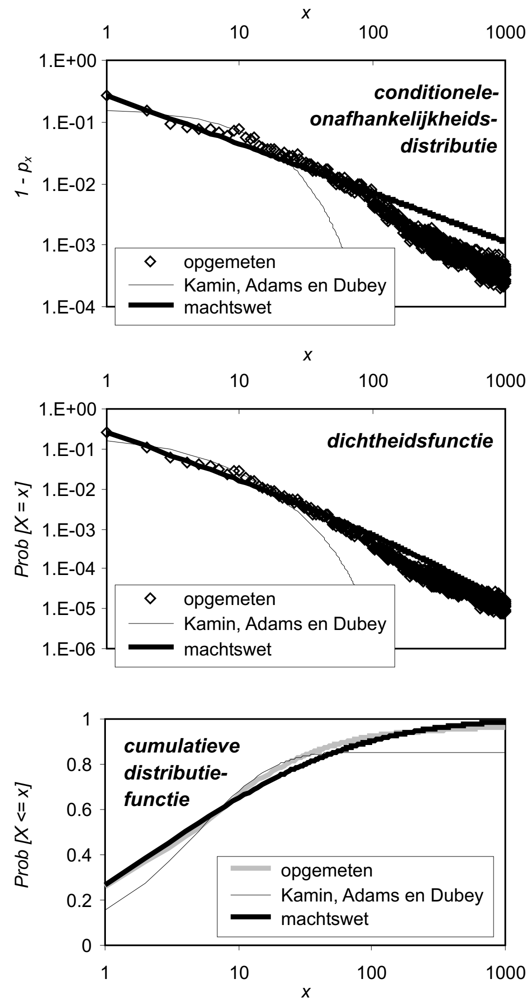
In deze doctoraatsthesis stellen we voor p_x te benaderen m.b.v. een machtswet:

$$p_x \approx 1 - \alpha x^{-\beta}. \quad (1.4)$$

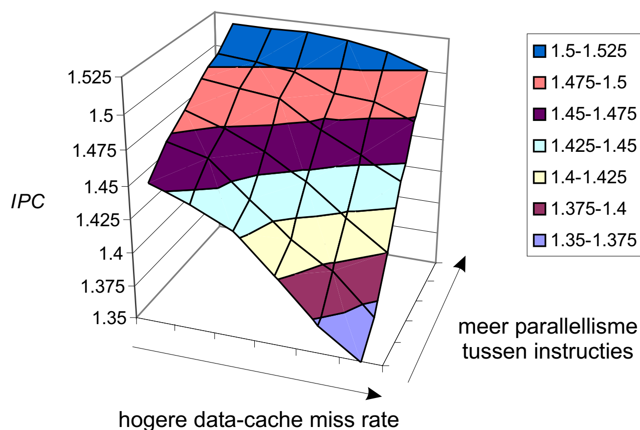
In figuur 1.7 wordt een voorbeeld gegeven van een dergelijke benadering voor de IBS instructiestroom real_gcc. Op deze figuur is duidelijk te zien dat de machtswet een betere benadering is dan de exponentiële functie. Een dergelijke vaststelling hebben we kunnen maken voor nagenoeg alle benchmarks die we geanalyseerd hebben, behalve voor li en compress. Voor deze twee benchmarks blijkt de exponentiële benadering nauwkeuriger te zijn dan de machtswet. Een mogelijke verklaring hiervoor is dat deze twee benchmarks het grootste deel van de uitvoeringstijd spenderen in een aantal kleine lussen waardoor er geen 'dikke staart' optreedt in de distributie en waardoor de exponentiële benadering bijgevolg beter blijkt te zijn.

Door het benaderen van de opgemeten distributies door de theoretische distributies worden de theoretische parameters α en β berekend. Deze twee parameters kunnen we vervolgens gebruiken in een analytisch model van een programma. Een dergelijk analytisch model bestaat dan uit een beperkt aantal parameters, een dertigtal in ons geval: 19 probabiliteiten voor de instructiemix, 2 parameters (α en β) voor de afhankelijkheden tussen instructies en 11 parameters voor het karakteriseren van het sprong- en cachegedrag. Dit analytisch model van een computerprogramma kan dan als statistisch profiel gebruikt worden bij het genereren van synthetische-instructiestromen. Op deze manier stellen we een hybride analytisch-statistische methodologie op voor het schatten van de prestatie in een vroeg ontwerpsstadium.

Vervolgens hebben we de absolute alsook de relatieve nauwkeurigheid geëvalueerd van deze hybride analytisch-statistische methode en is gebleken dat deze methode quasi even nauwkeurig is als statistische simulatie. Een belangrijk voordeel van een dergelijk analytisch model is dat deze methodologie aangewend kan worden om exploraties te doen in de ruimte van de computerprogramma's. Dit kan op een eenvoudige manier gebeuren door de verschillende parameters in het analytisch model te variëren. Op deze manier kan de volledige ruimte van applicaties geëxploreerd worden, dit in tegenstelling tot het



Figuur 1.7: Registerafhankelijkheden: schatting van de theoretische parameters α en β a.d.h.v. p_x voor de IBS instructiestroom real_gcc. De dichtheidsfunctie en de cumulatieve distributiefunctie worden eveneens weergegeven.



Figuur 1.8: Ruimte van computerprogramma's: IPC i.f.v. de datacache miss rate en het parallellisme tussen instructies.

beperkt aantal punten dat vertegenwoordigd wordt door een verzameling benchmarks. In figuur 1.8 wordt een dergelijk voorbeeld gegeven. In deze grafiek wordt de data-cache miss rate gevarieerd t.o.v. de structuur van de afhankelijkheden tussen instructies. Een dergelijke grafiek kan ons een aantal interessante inzichten aanreiken, b.v. dat door het parallellisme tussen instructies te verhogen, het prestatieverlies t.g.v. datacachemisses gedeeltelijk opgevangen kan worden.

5 Vermogenschatting

Vermogenverbruik is een bijzonder belangrijk criterium bij het ontwerp van hedendaagse microprocessors [3]. Voor draagbare toepassingen, zoals laptops, gsm's en digitale zakagenda's, is de levensduur van de batterij uiteraard een belangrijk gegeven. Ook voor systemen die bedoeld zijn voor een compleet ander segment van de markt, namelijk zogenaamde *high-end* systemen zoals b.v. werkstations en servers, is vermogenverbruik een belangrijk criterium. De steeds hogere klokfrequenties, het steeds toenemend aantal transistors op een chip en de steeds complexer wordende microarchitecturen leiden tot een sterke toename van het vermogenverbruik met een sterke toename van de temperatuur van de chip tot gevolg. Om een te grote opwarming te-

gen te gaan zal bijgevolg in een kostelijke verpakking en bijhorende koeling moeten voorzien worden wat de totale kostprijs per processor sterk doet toenemen.

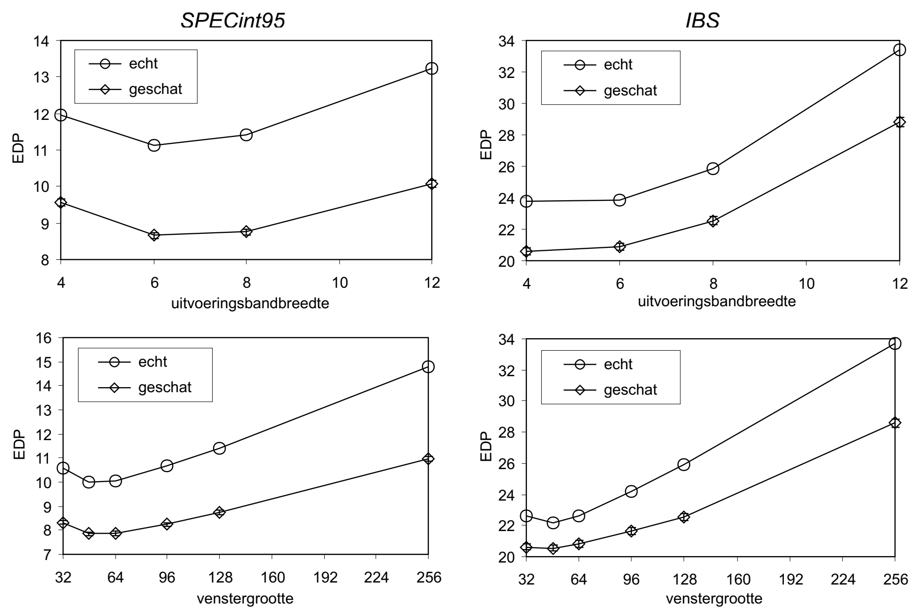
Opdat ontwerpers niet geconfronteerd zouden worden met een onaanvaardbaar vermogenverbruik in een laat stadium van het ontwerp, is het belangrijk om reeds in een vroeg ontwerpsstadium vermogen in beschouwing te nemen. In de literatuur zijn een aantal hulpmiddelen beschreven die hieraan tegemoet komen door vermogenverbruik te schatten op architecturaal niveau, b.v. Wattch [5], SimplePower [40, 41], PowerTimer [4] en TEM^{2}P^{2}EST [7]. Echter, al deze hulpmiddelen zijn gebaseerd op architecturale simulatie waardoor vermogenschatting op architecturaal niveau evenveel beperkt wordt door de simulatietijd als het geval is voor prestatieschatting. Hier kan aan tegemoet gekomen worden door de vermogenmodellen te integreren in een instructiestroomgedreven simulator en vervolgens statistische simulatie toe te passen [14]. Dit vormt dan ook het onderwerp van hoofdstuk 5 van de doctoraatsthesis.

In hoofdstuk 5 hebben we een bestaand vermogenmodel op architecturaal niveau, namelijk Wattch [5], geïntegreerd in onze simulator. Vervolgens hebben we de nauwkeurigheid geëvalueerd van statistische simulatie in de context van het schatten van vermogenverbruik. Om de nauwkeurigheid te evalueren hebben we de vermogenschatting opgemeten via simulatie van een synthetische-instructiestroom vergeleken met het vermogenverbruik dat opgemeten werd via simulatie van de echte instructiestroom. De absolute voorspellingsfout op het vermogen blijkt in alle onderzochte gevallen niet groter te zijn dan 10%. De relatieve nauwkeurigheid van statistische simulatie is bovendien ook bijzonder hoog.

Gebruikmakend van een schatting van de prestatie en een schatting van het vermogenverbruik van een computerprogramma, beide gegenereerd via statistische simulatie, kunnen we nu een schatting maken van de energie-efficiëntie van een microarchitectuur. Een veelgebruikte energie-efficiëntie metriek is het zogenaamde *energy-delay product* (EDP) [3] en wordt als volgt gedefinieerd:

$$\begin{aligned} EDP &= \frac{energieverbruik}{instructie} \cdot \frac{klokcycli}{instructie} \\ &= \left(\frac{1}{IPC}\right)^2 \cdot \frac{energieverbruik}{klokcyclus}. \end{aligned} \tag{1.5}$$

In figuur 1.9 wordt het EDP weergegeven als functie van de uitvoe-



Figuur 1.9: Echte en geschatte EDP i.f.v. het aantal te selecteren instructies per klokcyclus voor een processor met een instructievenster van 128 instructies (bovenste grafieken) en i.f.v. de grootte van het instructievenster voor een processor die 8 instructies kan selecteren per klokcyclus (onderste grafieken).

ringsbandbreedte (bovenste grafieken) en de grootte van het instructievenster (onderste grafieken). In alle grafieken is duidelijk te zien dat de echte EDP-curve goed benaderd wordt door de geschatte EDP-curve, relatief gesproken. De absolute afwijking tussen beide curves is hoofdzakelijk te wijten aan overschattingen van de IPC. In de doctoraatsthesis wordt ook geïllustreerd dat statistische simulatie zinvol is bij het nagaan van de interactie tussen het vermogenverbruik en programmakarakteristieken. Voor een uitgebreide bespreking hiervan verwijzen we naar de thesis zelf.

6 Ontwerp van een werklast

Zoals aangehaald in de inleiding is het belangrijk om in het begin van het ontwerpsproces van een microprocessor representatieve benchmarks met bijhorende inputs te selecteren. Deze moeten representatief zijn voor de omgeving waarin de te ontwerpen processor operationeel zal zijn. Het samenstellen van een dergelijke werklast bestaat eigenlijk uit twee componenten: (i) het selecteren van benchmarks en (ii) het selecteren van bijhorende inputs. Het is duidelijk dat het onmogelijk is een zeer groot aantal dergelijke koppels in de werklast op te nemen. Deze werklast zal immers gebruikt worden gedurende het ganse ontwerpsproces om het groot aantal simulaties aan te sturen. Bijgevolg kan een te grote werklast leiden tot veel te grote simulatietijden wat de ontwerpstijd aanzienlijk zou verlengen. In het ideale geval zouden we over een beperkte verzameling benchmarks met bijhorende inputs willen beschikken die een goed beeld schept van de volledige ruimte van computerprogramma's.

Conceptueel kan de volledige ruimte van computerprogramma's opgevat worden als een p-dimensionale ruimte waarbij p staat voor het totaal aantal relevante programmakarakteristieken. Deze programmakarakteristieken kwantificeren het gedrag van een programma met betrekking tot het spronggedrag, het cachegedrag, de hoeveelheid parallellisme op instructieniveau (*instruction-level parallelism* of ILP), enz. Het is duidelijk dat het voorstellen van de verschillende benchmark-input paren in een dergelijke p-dimensionale ruimte onmogelijk is. Bovendien bestaat er correlatie tussen de verschillende programmakarakteristieken wat de interpretatie van de voorstelling in deze p-dimensionale ruimte alleen maar bemoeilijkt. Daarom stellen we voor deze p-dimensionale ruimte te transformeren en te reduceren naar een q-dimensionale ruimte waarbij $q \ll p$ (typisch $q = 2$ tot $q = 4$). Bij

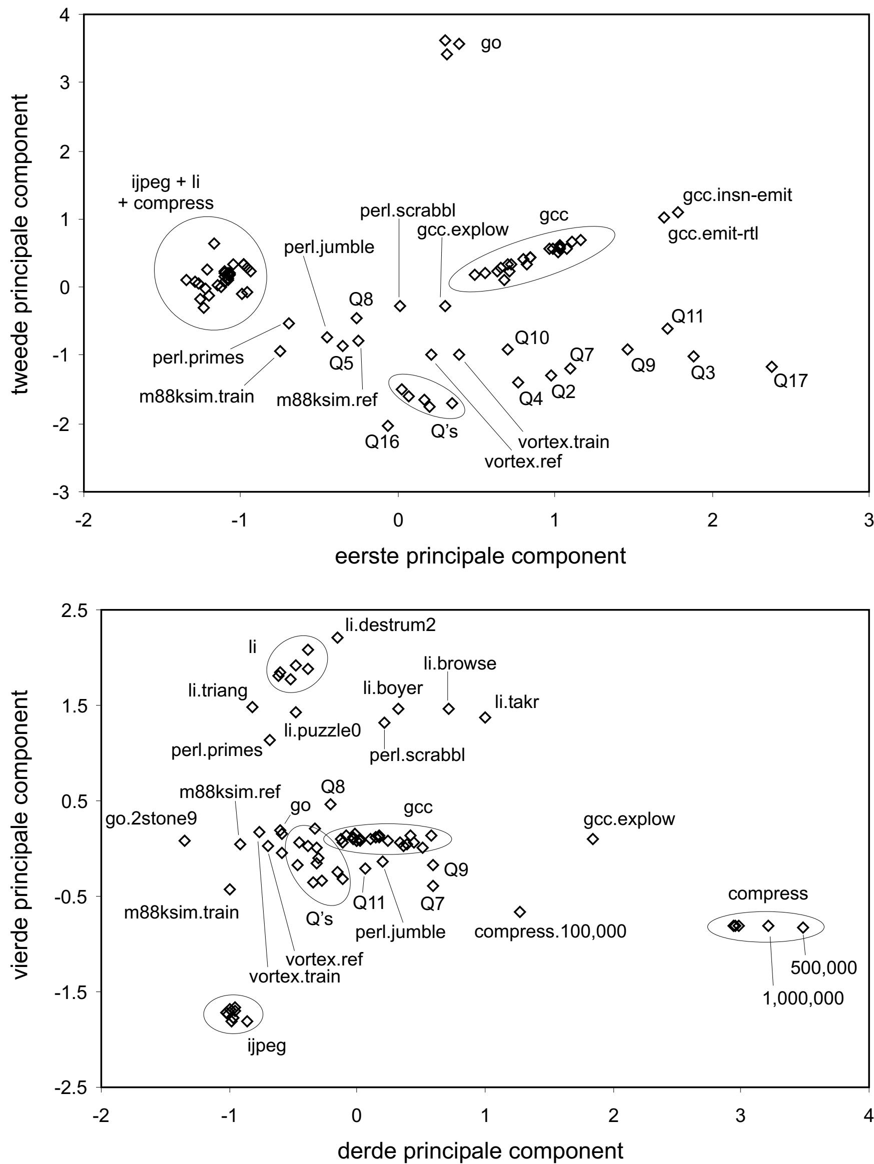
deze transformatie worden de oorspronkelijke programmakarakteristieken (die sterk gecorreleerd zijn) getransformeerd naar lineaire combinaties van de oorspronkelijke karakteristieken op een zodanige manier dat de nieuwe karakteristieken ongecorreleerd zijn. Van deze getransformeerde karakteristieken worden vervolgens diegene behouden die het meest informatie bevatten van de oorspronkelijke variabelen (in statistische termen: met de grootste variantie). Dit worden de principale componenten genoemd. Deze q karakteristieken spannen dan de gereduceerde q-dimensionale ruimte op. Een dergelijke transformatie wordt berekend via principale-componentenanalyse (PCA). Op deze manier wordt de interpretatie van de voorgestelde gegevens aanzienlijk vereenvoudigd omwille van twee redenen: (i) q is zeer klein en (ii) de assen in de q-dimensionale ruimte zijn ongecorreleerd.

Na deze transformatie kunnen de benchmark-input koppels voorgesteld worden in deze q-dimensionale ruimte. In figuur 1.10 wordt hiervan een voorbeeld gegeven voor de SPECint95 benchmarks (li, compress, gcc, go, perl, ijpeg, vortex en m88ksim) alsook voor een databankapplicatie die een reeks TPC-D vragen (*queries*) beantwoordt[4] (deze worden aangegeven door de letter Q gevolgd door het nummer van de vraag). Via clusteranalyse kunnen we vervolgens ook gelijkaardige of sterk verschillende benchmark-input koppels identificeren. Clusteranalyse is eigenlijk een iteratieve techniek die in iedere iteratie de punten die het dichtst bij elkaar liggen in de ruimte eerste gaat verbinden. Het middelpunt van het aldus gegenereerde 'cluster' vertegenwoordigt dan de zojuist verbonden punten in de volgende iteratie. Op deze manier wordt als het ware een netwerk van clusters gevormd waarbij de afstand tussen de verschillende clusters voorgesteld kan worden in een dendrogram, zie figuur 1.11. M.a.w. lange verbindingslijnen geven een grote afstand aan of dus in onze context een sterk verschillend gedrag; korte verbindingslijnen geven een korte afstand aan of dus een gelijkaardig gedrag.

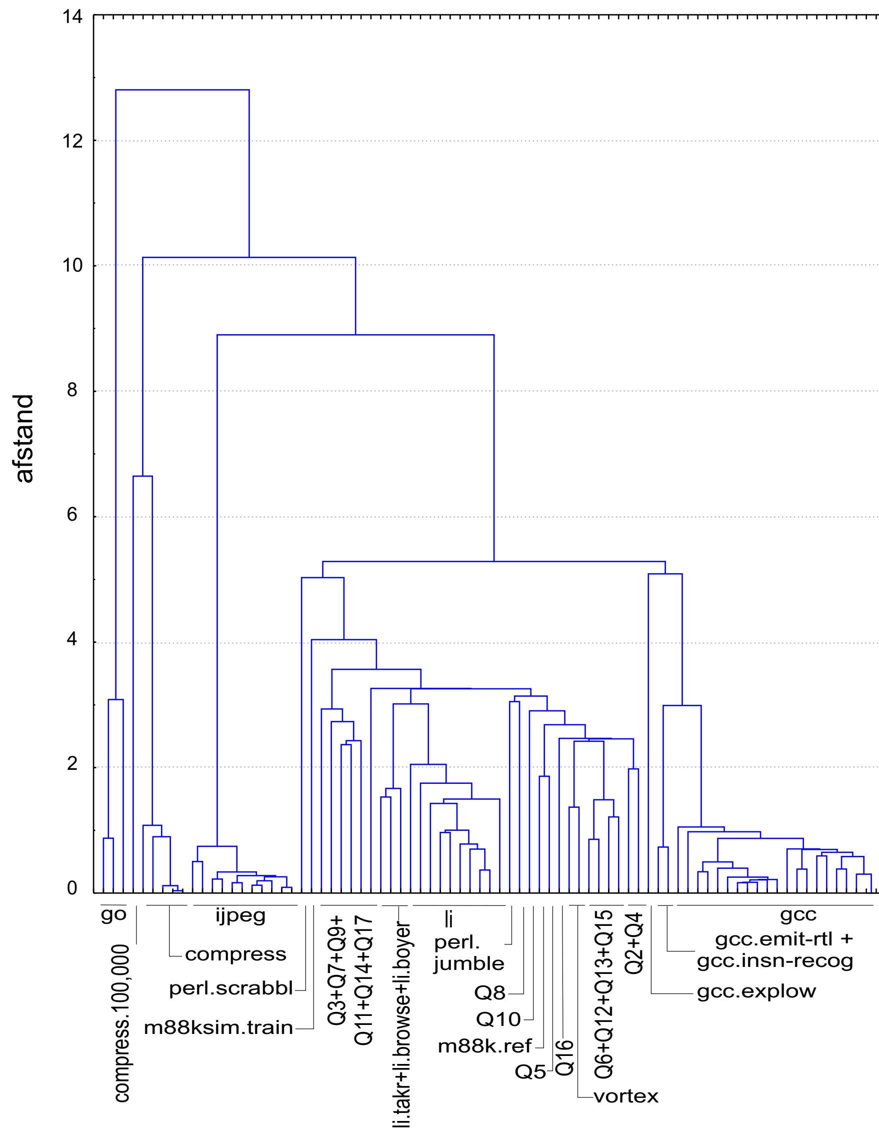
Deze twee voorstellingen van de ruimte van computerprogramma's, zie figuren 1.10 en 1.11, kunnen ons bijzonder nuttige informatie verschaffen betreffende de gelijkenissen en/of verschillen tussen verschillende benchmark-input koppels:

- Uit beide figuren blijkt duidelijk dat de benchmarks go, ijpeg en compress geïsoleerde punten zijn in deze ruimte. In het dendrogram dat gegenereerd wordt via clusteranalyse, zie figuur 1.11,

[4] `http://www.tpc.org`



Figuur 1.10: Ruimte van computerprogramma's opgespannen door de principale componenten: eerste component versus tweede component (bovenste grafiek) en derde versus vierde component (onderste grafiek).



Figuur 1.11: Clusteranalyse.

is immers duidelijk te zien dat deze drie benchmarks verbonden zijn met de andere benchmarks in de ruimte via lange verbindingslijnen. Via een gedetailleerde interpretatie van de principale componenten in termen van de oorspronkelijke variabelen kunnen we verklaringen geven voor dit bijzonder gedrag. Bijvoorbeeld, voor compress is dit duidelijk te wijten aan de hoge datacache miss rate; voor go daarentegen is dit te wijten aan een combinatie van factoren zoals de lage sprongvoorspellingsnauwkeurigheid, het percentage logische operaties en het hoog gehalte parallellisme op instructieniveau.

- Sommige benchmark-input koppels komen voor in sterke clusters. Dit is duidelijk te zien in een dendrogram wanneer dergelijke koppels verbonden zijn via korte verbindingslijnen. Dit geeft aan dat deze koppels een zeer vergelijkbaar gedrag vertonen. Bijgevolg kan een dergelijke cluster vertegenwoordigd worden door één enkel koppel of door een beperkt aantal koppels uit deze cluster. Op deze manier kan de totale simulatietijd in het ontwerpsproces aanzienlijk ingekort worden vermits er slechts een beperkt aantal benchmark-input koppels gesimuleerd moeten worden. Dit is bijvoorbeeld het geval voor een groot deel van de inputs geassocieerd met gcc, ijpeg en compress.

- De totale simulatietijd kan ook nog op een andere manier ingekort worden, namelijk door benchmark-input koppels te selecteren tijdens het samenstellen van een werklast met een zo klein mogelijk aantal dynamische instructies, of m.a.w. met een zo klein mogelijke dynamische instructiestroom. Bijvoorbeeld, voor vortex liggen de punten geassocieerd met inputs train en ref zeer dicht bij elkaar. Nochtans bevat de instructiestroom van de train input circa 3 miljard instructies terwijl de ref input circa 92 miljard instructies bevat. Bijgevolg zouden we kunnen opteren de train input te includeren in onze werklast i.p.v. de ref input wat de simulatietijd zou verkorten met een factor 30.

- Een dergelijke analyse geeft ons ook een unieke gelegenheid om de impact van een input op het gedrag van een programmauitvoering te identificeren. Zo heeft de input, of in dit geval de vraag die gesteld wordt aan de databankapplicatie, een grote invloed op het gedrag van de databankapplicatie. Dit valt af te leiden uit het feit dat de verschillende punten in de ruimte met label

Qx wijdverspreid voorkomen in de ruimte. Uit analyse van dit fenomeen blijkt dat de vraag die gesteld wordt aan de databankapplicatie een grote invloed heeft op het instructiecachegedrag en het spronggedrag.

In hoofdstuk 6 van de doctoraatsthesis wordt de hier voorgestelde techniek eveneens gevalideerd. Daarbij hebben we nagegaan of benchmark-input koppels die dicht bij elkaar liggen in de ruimte ook een gelijkaardig gedrag vertonen als functie van een aantal architecturale parameters, zoals de grootte van de cache, het type sprongvoorspeller, de grootte van het instructievenster, enz. De validatie blijkt deze stelling inderdaad te bevestigen.

7 Conclusie

Zoals aangegeven in de inleiding zijn er een aantal belangrijke aspecten verbonden met de vroegste ontwerpsstadia van een microprocessor. In wat volgt sommen we kort op hoe deze doctoraatsthesis hiertoe bijgedragen heeft. Ten eerste werd gesteld dat we over een werklast moeten beschikken die representatief is voor de operationele omgeving van de te ontwerpen microprocessor. In hoofdstuk 6 hebben we een nieuwe methodologie voorgesteld die gebaseerd is op multivariate data-analysetechnieken, die ons toelaat de invloed van de input van een computerprogramma op het gedrag ervan te analyseren. Deze techniek heeft als belangrijke toepassing dat hiermee gezocht kan worden naar representatieve benchmark-input koppels rekening houdend met de grootte van de instructiestroom. We wensen immers benchmark-input koppels te selecteren met een beperkte grootte van de instructiestroom (zonder daarbij aan nauwkeurigheid in te boeten gedurende de simulaties) teneinde de totale simulatietijd in te korten.

Ten tweede werd gesteld dat architecturale simulaties ontzettend veel tijd in beslag nemen tijdens het ontwerpsproces van een microprocessor. In deze doctoraatsthesis hebben we duidelijk geïllustreerd dat statistische simulatie hieraan kan tegemoet komen. Statistische simulatie is immers een snelle simulatietechniek (convergentie na een paar miljoen synthetische instructies) die bovendien vrij nauwkeurig is. De absolute voorspellingsfout op de IPC is in bijna alle gevallen kleiner dan 15% tot 20%. Daarnaast is ook gebleken uit dit onderzoek dat de relatieve nauwkeurigheid zeer goed is. Bovendien hebben we ook aangetoond dat om syntactische correctheid te kunnen garanderen van

de gegenereerde synthetische-instructiestromen we een terugkoppellus moeten implementeren in de synthetische-instructiestroomgenerator. We hebben ook aangetoond dat het modelleren van clusters van cache misses de nauwkeurigheid van statistische simulatie aanzienlijk kan verhogen. Een andere belangrijke bijdrage van dit werk is de vaststelling dat de distributie van de registerafhankelijkheden zich gedraagt volgens een machtswet, in tegenstelling tot de exponentiële benadering die in vroeger werk voorgesteld werd. Deze notie hebben we vervolgens aangewend om een hybride analytisch-statistische methodologie voor te stellen die bijna even nauwkeurig is als statistische simulatie. Deze nieuwe methodologie laat ons bovendien toe de ruimte van de computerprogramma's op een efficiënte manier te exploreren. Bijgevolg kunnen we concluderen dat statistische simulatie inderdaad bruikbaar is in een vroeg ontwerpsstadium. Statistische simulatie is b.v. in staat om op een efficiënte manier een regio aan te duiden in de ontwerpsruimte die dan verder geëvalueerd kan worden via gedetailleerde en dus trage architecturale simulaties gebruik makend van echte instructiestromen.

Ten derde werd gesteld dat ook technologische aspecten zoals b.v. vermogenverbruik opgenomen moeten worden in het begin van het ontwerpsproces zodat de computerontwerpers niet verrast zouden zijn door een onaanvaardbaar hoog vermogenverbruik in een later stadium van het ontwerp. In dit doctoraat hebben we aangetoond dat statistische simulatie zeer nauwkeurig is bij het schatten van vermogenverbruik met een absolute fout die steeds kleiner is dan 10%. Zoals gedemonstreerd kan statistische simulatie dus ook ingezet worden bij het zoeken naar energie-efficiënte microarchitecturen. Bovendien geeft deze methode ons een unieke kans om de invloed van programmakarakteristieken op het vermogenverbruik te onderzoeken.

Bibliografie

[1] T. Austin, E. Larson, en D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, Februari 2002.

[2] P. Bose en T. M. Conte. Performance analysis and its impact on design. *IEEE Computer*, 31(5):41–49, Mei 1998.

[3] D. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, en P. W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, November/December 2000.

[4] D. Brooks, M. Martonosi, J.-D. Wellman, en P. Bose. Power-performance modeling and tradeoff analysis for a high end microprocessor. In *Proceedings of the Power-Aware Computer Systems (PACS'00) held in conjunction with ASPLOS-IX*, November 2000.

[5] D. Brooks, V. Tiwari, en M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, blzn. 83–94, Juni 2000.

[6] R. Carl en J. E. Smith. Modeling superscalar processors via statistical simulation. In *Workshop on Performance Analysis and its Impact on Design (PAID-98), held in conjunction with the 25th Annual International Symposium on Computer Architecture (ISCA-25)*, Juni 1998.

[7] A. Dhodapkar, C. H. Lim, G. Cai, en W. R. Daasch. TEM^{2}P^{2}EST: A thermal enabled multi-model power/performance estimator. In *Proceedings of the Power-Aware Computer Systems (PACS'00) held in conjunction with ASPLOS-IX*, November 2000.

[8] P. K. Dubey, G. B. Adams III, en M. J. Flynn. Instruction window size trade-offs and characterization of program parallelism. *IEEE Transactions on Computers*, 43(4):431–442, April 1994.

[9] L. Eeckhout, T. Vander Aa, B. Goeman, H. Vandierendonck, R. Lauwereins, en K. De Bosschere. Application domains for fixed-length block structured architectures. In *Proceedings of the Sixth Australasian Computer Systems Architecture Conference (ACSAC 2001)*, blzn. 35–44, Januari 2001.

[10] L. Eeckhout en K. De Bosschere. Statistical simulation of superscalar architectures using commercial workloads. In *Fourth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-4) held in conjunction with the Seventh International Symposium on High Performance Computer Architecture (HPCA-7)*, Januari 2001.

[11] L. Eeckhout en K. De Bosschere. Quantifying behavioral differences between multimedia and general-purpose workloads. *Journal of Systems Architecture*, 2003. Aanvaard voor publicatie.

[12] L. Eeckhout, K. De Bosschere, en H. Neefs. On the feasibility of fixed-length block structured architectures. In *Proceedings of the 5th Australasian Computer Architecture Conference ACAC 2000*, blzn. 17–25, Januari 2000.

[13] L. Eeckhout en K. De Bosschere. Nonuniform behavior in instruction traces for contemporary processors. In *AIP Conference Proceedings of the Fourth International Conference on Computing Anticipatory Systems (CASYS'2000)*, blzn. 662–672, Augustus 2000.

[14] L. Eeckhout en K. De Bosschere. Early design phase power/performance modeling through statistical simulation. In *Proceedings of the 2001 International IEEE Symposium on Performance Analysis of Systems and Software (ISPASS-2001)*, blzn. 10–17, November 2001.

[15] L. Eeckhout en K. De Bosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*, blzn. 25–34, September 2001.

[16] L. Eeckhout en K. De Bosschere. Increasing the accuracy of statistical simulation for modeling superscalar processors. In *The 20th IEEE International Performance, Computing and Communications Conference (IPCCC 2001)*, blzn. 196–204, April 2001.

[17] L. Eeckhout, K. De Bosschere, en H. Neefs. Performance analysis through synthetic trace generation. In *The IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2000)*, blzn. 1–6, April 2000.

[18] L. Eeckhout, H. Neefs, en K. De Bosschere. Estimating IPC of a block structured instruction set architecture in an early design stage. In *Parallel Computing: Fundamentals and Applications; Proceedings of the International Conference ParCo99*, blzn. 468–475, Januari 2000.

[19] L. Eeckhout, H. Neefs, K. De Bosschere, en J. Van Campenhout. Improving loop performance on a block structured architecture through predication. In *Proceedings of the Tenth International Conference on Parallel and Distributed Computing and Systems*, blzn. 457–462, Oktober 1998.

[20] L. Eeckhout, H. Neefs, K. De Bosschere, en J. Van Campenhout. Investigating the implementation of a block structured processor architecture in an early design stage. In *Proceedings of the 25th Euromicro Conference*, volume 1, blzn. 186–193, September 1999.

[21] L. Eeckhout, H. Neefs, K. De Bosschere, en J. Van Campenhout. On the organisation and implementation of a fixed-length block structured instruction set architecture. In *Proceedings of the 1999 ACM/IEEE International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, blzn. 77–82, Maart 1999.

[22] L. Eeckhout, H. Neefs, en K. De Bosschere. Early design stage exploration of fixed-length block structured architectures. *Journal of Systems Architecture*, 46:1469–1486, December 2000.

[23] L. Eeckhout, H. Vandierendonck, en K. De Bosschere. Designing workloads for computer architecture research. *IEEE Computer*, 2003. Aanvaard voor publicatie.

[24] L. Eeckhout, H. Vandierendonck, en K. De Bosschere. Quantifying the impact of input data sets on program behavior and

its applications. *Journal of Instruction-Level Parallelism*, 2003. `http://www.jilp.org`. Conditioneel aanvaard voor publicatie.

[25] L. Eeckhout, H. Vandierendonck, en K. De Bosschere. How input data sets change program behaviour. In *Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-02) held in conjunction with the Eighth International Symposium on High-Performance Computer Architecture (HPCA-8)*, Februari 2002.

[26] L. Eeckhout, H. Vandierendonck, en K. De Bosschere. Workload design: Selecting representative program-input pairs. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT-2002)*, blzn. 83–94, September 2002.

[27] M. J. Flynn, P. Hung, en K. W. Rudd. Deep-submicron microprocessor design issues. *IEEE Micro*, 19(4):11–22, Juli/Augustus 1999.

[28] M. K. Gowan, L. L. Biro, en D. B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. In *Proceedings of the 35th Design Automation Conference (DAC-1998)*, blzn. 726–731, Juni 1998.

[29] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, en P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q1 2001.

[30] R. A. Kamin III, G. B. Adams III, en P. K. Dubey. Dynamic trace analysis for analytic modeling of superscalar performance. *Performance Evaluation*, 19(2-3):259–276, Maart 1994.

[31] R. E. Kessler, E. J. McLellan, en D. A. Webb. The Alpha 21264 microprocessor architecture. In *Proceedings of the 1998 International Conference on Computer Design (ICCD-98)*, blzn. 90–95, Oktober 1998.

[32] S. Melvin en Y. Patt. Enhancing instruction scheduling with a block-structured ISA. *International Journal of Parallel Programming*, 23(3):221–243, 1995.

[33] H. Neefs. A preliminary study of a fixed length block structured instruction set architecture. Technical Report Paris 96-07, Dept. of Electronics and Information Systems, Ghent University, November 1996.

[34] D. B. Noonburg en J. P. Shen. Theoretical modeling of superscalar processor performance. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*, blzn. 52–62, November 1994.

[35] S. Nussbaum en J. E. Smith. Modeling superscalar processors via statistical simulation. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*, blzn. 15–24, September 2001.

[36] S. Nussbaum en J. E. Smith. Statistical simulation of symmetric multiprocessor systems. In *Proceedings of the 35th Annual Simulation Symposium 2002*, blzn. 89–97, April 2002.

[37] M. Oskin, F. T. Chong, en M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, blzn. 71–82, Juni 2000.

[38] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, en J. Emer. Instruction fetching: Coping with code bloat. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA-22)*, blzn. 345–356, Juni 1995.

[39] T. Vander Aa, L. Eeckhout, B. Goeman, H. Vandierendonck, T. Van Achteren, R. Lauwereins, en K. De Bosschere. Optimizing a 3D image reconstruction algorithm: Investigating the interaction between the high-level implementation, the compiler and the architecture. In *Proceedings of the Seventh Asia-Pacific Computer Systems Architecture Conference (ACSAC-2002)*, blzn. 119–126, Februari 2002.

[40] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, en W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, blzn. 95–106, Juni 2000.

[41] W. Ye, N. Vijaykrishnan, M. Kandemir, en M. J. Irwin. The design and use of SimplePower: A cycle-accurate energy estimation tool. In *Proceedings of the 37th Design Automation Conference*, blzn. 340–345, Juni 2000.

[42] K. C. Yeager. MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2), April 1996.

Accurate Statistical Workload Modeling

Contents

1 Introduction **1**
1.1 Microprocessor design 2
1.2 Contributions 5
1.3 Overview 6

2 Statistical simulation **9**
2.1 Out-of-order architecture 9
2.2 Statistical simulation 13
2.3 Applications 15
2.4 Statistical profiling 17
2.4.1 Microarchitecture-independent characteristics . . 18
2.4.2 Microarchitecture-dependent characteristics . . . 20
2.5 Synthetic trace generation and simulation 21
2.6 Methodology 26
2.7 Performance prediction accuracy 27
2.7.1 Absolute accuracy 30
2.7.2 Relative accuracy 40
2.8 Simulation speed 46
2.9 Summary 47

3 Increasing the accuracy of statistical simulation **49**
3.1 Higher-order ILP distributions 49
3.2 Clustered cache misses 52
3.2.1 Clustered I-cache misses 54
3.2.2 Clustered D-cache misses 57
3.2.3 Overall prediction accuracy 60
3.3 Related work 65
3.3.1 Statistical simulation 65
3.3.2 Analytical modeling 68
3.3.3 Trace sampling 69

3.3.4 Other approaches 70
3.4 Summary 73

4 Register traffic characteristics 75
4.1 Register traffic characteristics 75
4.2 Distribution fitting 79
4.3 Hybrid analytical-statistical modeling 86
4.3.1 Analytical workload model 86
4.3.2 Evaluation 87
4.4 Workload space exploration 89
4.5 Summary 93

5 Power modeling 95
5.1 Introduction 95
5.2 Power modeling 96
5.2.1 General concepts 96
5.2.2 Architectural power modeling 97
5.2.3 Power/performance metrics 99
5.3 Evaluation 100
5.3.1 Absolute accuracy 100
5.3.2 Relative accuracy 103
5.3.3 Energy efficiency 105
5.4 Energy behavior vs. program characteristics 107
5.5 Summary 112

6 Workload design 113
6.1 Introduction 114
6.2 Workload characterization 115
6.3 Data analysis 117
6.3.1 Principal components analysis 117
6.3.2 Cluster analysis 119
6.3.3 Workload analysis 119
6.4 Evaluation 121
6.4.1 Experimental setup 124
6.4.2 Results 124
6.4.3 Preliminary validation 135
6.5 Related work 138
6.6 Conclusion 141

7 Conclusion **143**
7.1 Summary 143
7.2 Future work 145

Bibliografy **149**

List of Figures

2.1 Out-of-order architecture 10
2.2 Statistical simulation: framework 13
2.3 Program characteristic determination 21
2.4 Synthetic trace generation 22
2.5 Deviation between desired and realized distribution . . . 25
2.6 Feedback loop in synthetic trace generator 26
2.7 Accuracy when modeling ILP: instruction mix and inter-operation dependencies 30
2.8 Accuracy when modeling branch behavior 32
2.9 Accuracy when modeling I-cache behavior: 'small' cache 32
2.10 Accuracy when modeling I-cache behavior: 'large' cache 33
2.11 Accuracy when modeling D-cache behavior: 'small' cache 34
2.12 Accuracy when modeling D-cache behavior: 'large' cache 35
2.13 Overall prediction accuracy: 'small' cache 36
2.14 Overall prediction accuracy: 'large' cache 36
2.15 Absolute accuracy: impact of characteristics 38
2.16 Relative accuracy: issue width 41
2.17 Relative accuracy: window size 42
2.18 Relative accuracy: fetch width 43
2.19 Relative accuracy: branch misprediction penalty 44
2.20 Relative accuracy: load latency 45
2.21 Jitter of statistical simulation 46

3.1 Accuracy using higher-order ILP distributions 51
3.2 Modeling bursty instruction cache behavior 53
3.3 Enhanced modeling of I-cache behavior: 'small' cache . . 55
3.4 Enhanced modeling of I-cache behavior: 'large' cache . . 56
3.5 Impact on performance of delayed hits 57
3.6 Enhanced modeling of D-cache behavior: 'small' cache . 58
3.7 Enhanced modeling of D-cache behavior: 'large' cache . 59

3.8 Overall prediction accuracy: 'small' cache 61
3.9 Overall prediction accuracy: 'large' cache 62
3.10 Relative accuracy when using enhanced cache statistics: fetch width 63

4.1 Register traffic characteristics: example 76
4.2 Register traffic characteristics for the SPECint95 benchmarks 77
4.3 Register traffic characteristics for the IBS traces 78
4.4 Register traffic characteristics: distribution fitting 82
4.5 Age of register instances for the SPECint95 benchmarks . 83
4.6 Age of register instances for the IBS traces 84
4.7 Absolute accuracy hybrid analytical-statistical model . . 88
4.8 Relative accuracy hybrid analytical-statistical model . . . 90
4.9 IPC as a function of α and β 91
4.10 IPC as a function of α, β, L1 and L2 D-cache miss rate . . 92

5.1 Power modeling for SPECint95: absolute accuracy 101
5.2 Power modeling for IBS: absolute accuracy 102
5.3 Power modeling: relative accuracy 104
5.4 EDP: relative accuracy 105
5.5 ED^{2}P: relative accuracy 106
5.6 IPC and energy per cycle as a function of I- and D-cache miss rate 109
5.7 IPC and energy per cycle as a function of branch prediction accuracy 110
5.8 Front end and back end activity on a branch misprediction 111

6.1 Factor loadings for gcc 126
6.2 Workload space for gcc 126
6.3 Factor loadings for postgres running the TPC-D queries . 127
6.4 Workload space for postgres running the TPC-D queries 128
6.5 Factor loadings for all program-input pairs 129
6.6 Workload space for all program-input pairs 130
6.7 Cluster analysis 131
6.8 Validation: I-cache behavior 136
6.9 Validation: D-cache behavior 136
6.10 Validation: branch predictor 137
6.11 Validation: window size and issue width 137

List of Tables

2.1 Out-of-order architecture 12
2.2 Benchmarks used 27
2.3 SPECint95 traces 28
2.4 IBS traces 29

3.1 Enhanced cache statistics: prediction accuracy 64

4.1 Fitted α and β values 85

6.1 Workload design: benchmarks part one 122
6.2 Workload design: benchmarks part two 123

Chapter 1

Introduction

The beginning is the half of every action.
Greek proverb

Moore's law states that the number of transistors that can be integrated on a chip doubles every 18 months. This empirical law can also be translated to the performance of microprocessor chips: microprocessor performance (measured as throughput of work) doubles every 18 months. This performance boost is due to a number of important improvements that have been realized over the years. First, compiler writers nowadays are able to generate highly optimized executables. Second, computer architects are designing more and more complex microarchitectures in order to get the highest possible performance in a given chip technology. A number of important microarchitectural features have been added to increase performance: branch prediction, speculative execution, memory disambiguation, prefetching, cache line prediction, trace caches, etc. Third, chip technology has improved dramatically so that nowadays several hundreds of millions of transistors can be integrated on a single chip, gigahertz clock frequencies can be obtained, etc.

The downside of this phenomenon, is the ever increasing complexity of designing such microprocessor systems, which translates into an increased design time and thus an increased time-to-market. A significant part of this complexity is caused by the fact that the technology increasingly impacts the microarchitectural design. For example, interconnect delays and power consumption are becoming key design issues, even at the microarchitectural level. Dealing with these issues is key for todays and future microprocessor design

methodologies. Before going into detail how this dissertation contributes to this research area, we first give a view on the microprocessor design process.

1.1 Microprocessor design

Designing a new microprocessor typically takes several years from the first ideas to the final fabricated chip. Generally, this design process consists of a number of steps [11, 12, 110]: workload composition, design space exploration, high level architectural definition and simulation, low level modeling and simulation and finally validation of the various abstraction levels. We give special attention to the first three steps since these steps are most closely related to the subject of this dissertation.

In the first step, a workload is composed which means that a number of benchmarks need to be chosen that are representative for the target domain of operation of the microprocessor [64]. For example, a representative workload for a microprocessor that is targeted for the desktop market will typically consist of a number of desktop applications such as a word processor, a spreadsheet, etc. For a workstation aimed at scientific research on the other hand, a representative workload should consist of a number of applications that are computation intensive, e.g., weather prediction, solving partial differential equations, etc. Embedded microprocessors should be designed with a workload consisting of digital signal processing (DSP) and multimedia applications. Note that composing a workload consists of two issues: (i) which benchmarks need to be chosen and (ii) which input data sets need to be selected so that representative program-input pairs are selected for the workload. It is important to realize that this design step is extremely crucial since the complete design process will be based on this workload. If the workload is badly composed, the microprocessor will be optimized for a workload that is not representative for the real workload. As such, the microprocessor might attain non-optimal performance in its target domain of operation.

In the second step, lead computer architects bound the space of potential designs. This is done together with experts from other disciplines, e.g., technology experts. As such, technology issues—interconnect delays, total chip area, power consumption, pin count, packaging cost, etc.—are incorporated in the earliest stages of the design. Note that this is extremely important for current and near

future chip technologies [7]. In these chip technologies, the interconnect delay does not scale with feature size [54]. As such, computer architects should avoid designing a microarchitecture that needs long wires throughout the chip since these long wires will limit the overall clock frequency. For example, early floorplanners [95] could be used to detect long wires in early stages of the design. Several propositions have been made in the recent literature to reduce the interconnects on the chip [51, 107, 109, 113, 121]. This aspect of technology is yet visible in commercial designs. In the Alpha 21264 [60] for example, the processor core has a clustered design to be able to clock the microprocessor at its target frequency. Another key design issue that is related to technology, is power consumption [13]. As microarchitectures are becoming increasingly complex and as chips are being clocked at ever higher frequencies, power consumption increases dramatically which increases the packaging cost and the cooling cost. As such, computer architects should consider power consumption in the earliest stages of the design in order not to be surprised with an unacceptable power consumption in the latest stages. In conclusion, computer architects should be able to make appropriate decisions in early stages of the design that consider technology aspects, such as interconnect delay and power consumption.

In the third step, a specific microarchitecture is chosen based on simulation results. Note that these simulations are done at a high level using an architectural simulator that is written in a high level programming language such as C, C++ or Java. Usually, companies have their own simulation infrastructure, such as Asim [49, 111] used by the Compaq design team now with Intel, and MET [93, 94] used by IBM. Virtutech released Simics [86], a platform for full system simulation. The SimpleScalar Tool Set [1, 18] is an architectural simulator that is widely used in the academia and the industry for evaluating uniprocessor microarchitectures at the architectural level. Other examples of architectural simulators developed by the academia are Rsim [70] at Rice University (for simulating shared-memory multiprocessors), SimOS [112] at Stanford University (for full system simulation of shared-memory multiprocessors), fMW [3] at Carnegie Mellon University (quite similar to SimpleScalar), and TFsim [89] at the University of Wisconsin–Madison (a full system multiprocessor simulator). Note that these simulators do model microarchitectures at a high level which introduces inaccuracies in the performance results when compared to real hardware [6, 27, 58]. It is also interesting to note that there exist two types of

architectural simulators: trace-driven and execution-driven. The basic difference between both is that an execution-driven simulator actually executes the executable as it would be executed on a real system—this involves interpreting the individual instructions. A trace-driven simulator on the other hand, only imposes dependencies between instructions since the instructions do not need to be re-executed—the exact sequence of instructions is already available in the trace. This gives a potential speed advantage for the trace-driven simulator at the cost of accuracy due to not simulating misspeculated paths [4, 5, 22]. Another disadvantage of trace-driven simulation is the fact that traces need to be stored on a hard drive, which might be impractical in case of long traces.

These simulation runs, even if they model a microarchitecture at a high level, are extremely time consuming. For example, simulating one second of a 2 GHz microprocessor at a typical speed of 50,000 cycles per second [9]—scale factor of 40,000X—already takes 11 hours of simulation. If we take into account that (i) several program-input pairs in the workload need to be simulated, (ii) per program-input pair several billions of instructions need to be simulated in order to be representative for the real workload, (iii) a huge amount of microarchitectural design points need to be evaluated, (iv) technology aspects need to be incorporated in the analysis, we could end up with a total simulation time of several months, if not years. Obviously, this is impractical as it will ultimately enlarge the time-to-market. As such, it is very important to have methods that could speed up this simulation process.

In the fourth step, these high level descriptions of the microarchitecture are translated to lower level descriptions: the register transfer level (RTL), the logic level, the gate level or the circuit level. The correctness of these various levels of description are verified in the fifth step [8].

In conclusion, we can state that there are a number of major issues involved concerning the earliest stages of the design:

- representative workloads: program-input pairs should be carefully chosen when designing a workload.

- fast simulation techniques: the architectural simulation process should be sped up.

- impact of technology: aspects such as interconnect delay and power consumption should be taken into account.

- accurate estimation: while searching for fast early design stage methods, the accuracy should be reasonable so that appropriate design decisions are made.

In this context, it is important to realize that making correct design decisions in the earliest stages of the design cycle can significantly reduce the time-to-market as well as the total design cost.

1.2 Contributions

This dissertation makes the following contributions to these early design stage issues:

- we evaluate statistical simulation as a viable early design stage method. The basic idea of statistical simulation is very simple: measure a number of characteristics of a program execution, generate a synthetic trace using these characteristics and simulate this synthetic trace. Due to the statistical nature of the technique and the fact that simple characteristics are used, performance characteristics quickly converge during simulation—the performance characteristics fluctuate as long as not all the program characteristics are realized sufficiently in the synthetic trace. Due to the use of simple characteristics, As such, we can conlude that statistical simulation indeed is a fast simulation technique. In its evaluation we consider both absolute and relative accuracy as we believe that relative accuracy is even more important than absolute accuracy in early stages of the design.

- we improve the accuracy of statistical simulation by proposing several improvements [32, 38, 39]: (i) higher-order inter-operation dependency distributions, (ii) the implementation of a feedback loop in the synthetic trace generator so that syntactical correctness can be guaranteed, and (iii) the modeling of clustered cache misses.

- we show that inter-operation dependency distributions generally exhibit power-law properties [37]. This observation is used to present a hybrid analytical-statistical methodology that is nearly as accurate as statistical simulation. In addition, this allows us to do workload space explorations by varying the various parameters in the workload model.

- we show that statistical simulation can be used for early design stage power modeling [36]. This is done by integrating an architectural power model in the synthetic trace simulator. Besides the evaluation of the accuracy of this approach, we also present the applicability of this method for getting insight in the impact of program characteristics on power consumption.

- finally, we propose a method for selecting representative program-input pairs while composing a workload [45, 46, 47, 48]. This method is based on multivariate statistical data analysis techniques.

Besides these contributions, other research was done during the last four years on a number of subjects. First, we applied statistical simulation for evaluating an experimental microarchitecture, namely a fixed-length block structured architecture [96], which is a particular form of the block structured architecture first proposed by Melvin and Patt [91]. The work that was done in this context evaluates this experimental architecture using statistical simulation and early design stage cycle time estimations [31, 34, 40, 41, 42, 43, 44]. Second, a number of subjects in the area of workload characterization were addressed: analysis of the non-uniform behavior of program executions [35] and quantifying behavioral differences between multimedia and general-purpose workloads [33, 132].

1.3 Overview

This dissertation is organized as follows. In chapter 2, we present the statistical simulation methodology. We detail on the statistical profiling step, the synthetic trace generation algorithm and the synthetic trace simulation process. We evaluate the modeling of the various components: the instruction-level parallelism (ILP), the branch behavior and the cache behavior. And we quantify the errors introduced by each of these. In the evaluation of the statistical simulation method, we consider absolute accuracy as well as relative accuracy. We also show that statistical simulation indeed is a fast simulation technique. In addition, we show that incorporating a feedback loop in the synthetic trace generator allows us to generate syntactically correct traces while preserving the representativeness.

Chapter 3 deals with increasing the accuracy of statistical simulation. We consider two possible ways for doing so: the use of higher-order ILP distributions and the modeling of clustered cache misses. At the end of chapter 3, we extensively discuss related work in the field of early design stage performance estimation and fast simulation techniques.

Chapter 4 builds on the observation that register traffic characteristics exhibit power-law properties. Indeed, the probability that an operation is dependent on an operation that is ahead in the dynamic instruction trace has a power-law distribution as a function of the distance between those two operations. The parameters α and β of the theoretical distribution are estimated through distribution fitting. These parameters are then used in a hybrid analytical-statistical model that is nearly as accurate as the statistical simulation methodology considered in chapters 2 and 3. An important application for the analytical workload model is that workload space explorations can be easily done by varying the various parameters in the model.

Chapter 5 deals with another interesting application of statistical simulation, namely early design stage power modeling. By integrating an architectural power estimator in the synthetic trace simulator, power can be estimated using synthetic traces. As such, accurate power estimates are obtained in early stages of the design cycle. Both the absolute and the relative accuracy are discussed. In addition, we discuss two applications of this approach: determining energy-efficient microarchitectures and investigating the interaction between the energy consumption per cycle and program characteristics, such as cache miss rate, branch misprediction rate, etc.

In chapter 6, we present a methodology that is based on statistical data analysis (principal components analysis and cluster analysis) to select representative program-input pairs. These analysis techniques provide us a better view on the workload space, i.e., the impact of input sets on program behavior can be measured. As such, representative inputs can be chosen to guide the microprocessor design process.

Finally, we conclude and discuss future research directions in chapter 7.

Chapter 2

Statistical simulation

Prediction is difficult, especially of the future.
Niels Bohr

In this chapter, we present the statistical simulation methodology. We discuss the three basic steps of this method: (i) statistical profiling, i.e., measuring program characteristics, (ii) synthetic trace generation using this statistical profile, and (iii) synthetic trace simulation. We extensively evaluate the accuracy of statistical simulation by measuring the prediction accuracy of the various characteristics in the statistical profile. Besides the absolute accuracy, we also quantify the relative accuracy as a function of various microarchitectural parameters. We also detail on the possible applications of statistical simulation. Finally, we show that statistical simulation indeed is a fast simulation methodology.

2.1 Out-of-order architecture

To validate statistical simulation, an out-of-order superscalar architecture[1] [64, 119] was assumed which is an architectural paradigm that is implemented in most contemporary microprocessors, such as the Alpha 21264 [77], Pentium 4 [66], MIPS R10000 [139], etc. In an out-of-order architecture, see Figure 2.1, instructions are fetched from the

[1] Many of the basic mechanisms of an out-of-order architecture were proposed by Tomasulo [130] for the design of the IBM 360 model 91 in 1967!

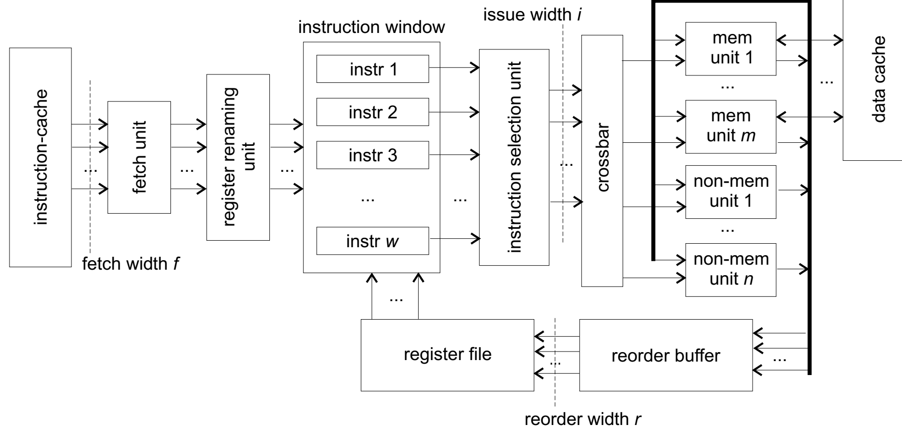


Figure 2.1: Out-of-order architecture: general view.

instruction cache (I-cache), on which register renaming is performed. Register renaming eliminates write-after-read (WAR) and write-after-write (WAW) dependencies from the instruction stream; only real read-after-write (RAW) data dependencies remain.[2] Once the instructions are transformed into a static single assignment form, they are dispatched to the *instruction window*, where the instructions wait for their source operands to become available (data-flow execution). Each clock cycle, ready instructions are selected from the instruction window to be executed on a functional unit. The number of instructions that can be selected for execution in one clock cycle, is restricted to the *issue width*. Further, bypassing is implemented which means that data-dependent instructions can be executed in consecutive cycles. Once an instruction is executed, the instruction can be retired, i.e., removed from the processor core and its results written to the register file or memory, when all previous instructions from the sequential instruction stream are retired. The number of instructions that can be retired in one clock cycle, is restricted to the *reorder width*.

Note that the organization described here is only one possible implementation of an out-of-order architecture in which one central structure, called the instruction window in this dissertation, serves for issuing instructions as well as for retiring instructions. Other organiza-

[2] In this dissertation, we assume perfect register renaming. In practice, this means we assume enough physical registers to rename all the instructions residing in the processor core.

tions exist in which two structures are implemented, the issue window (from which instructions are issued) and the reorder buffer (ROB). In the latter organization, the issue window can be made smaller simplifying the issue logic since the issue window collapses as instructions get executed. However, choosing one or the other organization does not impact the applicability of the statistical simulation methodology as presented in this dissertation.

A dynamic memory disambiguation strategy [56, 97] is implemented in the architectures which allows out-of-order execution of memory operations. A store instruction is issued when its source operands are available and writes its temporary result into a store buffer; that value is then written to memory when the corresponding store retires. A load accesses the L1 data cache (D-cache) and the store buffer in parallel. When the value is found in the store buffer, the value is returned from the store buffer; if this is not the case, the value is read from the L1 D-cache. Re-execution is implemented to recover from misspeculated loads, which re-executes the instructions that are dependent (directly or indirectly) on the misspeculated load.

The simulator that is used in this dissertation is a trace-driven simulator with comparable functionality as the SimpleScalar out-of-order simulator[3] [1, 18]. The parameters involved in our trace-driven simulator are tabulated in Table 2.1. In addition, the dimension of each parameter is given as it is configured in the experiments. In this dissertation, we will often refer to a w/i processor configuration which is an i-wide processor with an instruction window of w-entries. The fetch width and the reorder width are chosen to be the same as the issue width unless stated otherwise.

The reason why we chose wide-resource machines in our evaluation is that on such microarchitectures performance is more limited by program parallelism than by machine parallelism. This way, the capability of the statistical simulation methodology for modeling program parallelism is stressed appropriately. Consequently, if the technique is accurate for wide-resource machines, we can expect that the technique will also be useful for processors with less machine parallelism. The data presented in the evaluation section of this chapter confirm that the prediction errors are smaller for small-resource machines than for wide-resource machines in general. As such, we can conclude that

[3]The instruction window as described in this section corresponds to the register update unit (RUU) used by SimpleScalar.

Microarchitecture	
fetch width f	4 to 12
window size w	32 to 256
issue width i	4 to 12
number of non-memory units n	3 to 8
number of memory units m	2 to 6
reorder width r	4 to 12
front-end pipeline	4 stages: fetch, decode, rename and dispatch
Execution latencies	
integer	1 cycle
load	3 cycles (address calculation plus L1 data cache access)
multiply	8 cycles
floating-point	4 cycles
single precision divide	18 cycles (non-pipelined)
double precision divide	31 cycles (non-pipelined)
Branch prediction	
branch predictor	4K-entry meta predictor choosing between a 4K-entry bimodal predictor and an 8-bit gshare predictor indexing a 4K-entry table [90]
branch target buffer (BTB)	512 sets and 4-way set-associative
return address stack (RAS)	8 entries
Memory hierarchy	
instruction cache (I-cache)	'small': 8KB direct-mapped L1 'large': 32KB direct-mapped L1 32-byte blocks access time: 1 cycle
data cache (D-cache)	'small': 8KB direct-mapped L1 'large': 64KB 2-way set-associative L1 32-byte blocks access time: 2 to 4 cycles
L2 unified cache	'small': 64KB 2-way set-associative L2 'large': 256KB 4-way set-associative L2 32-byte blocks access time: 10 cycles
main memory	access time: 80 cycles

Table 2.1: The out-of-order architecture configurations used.

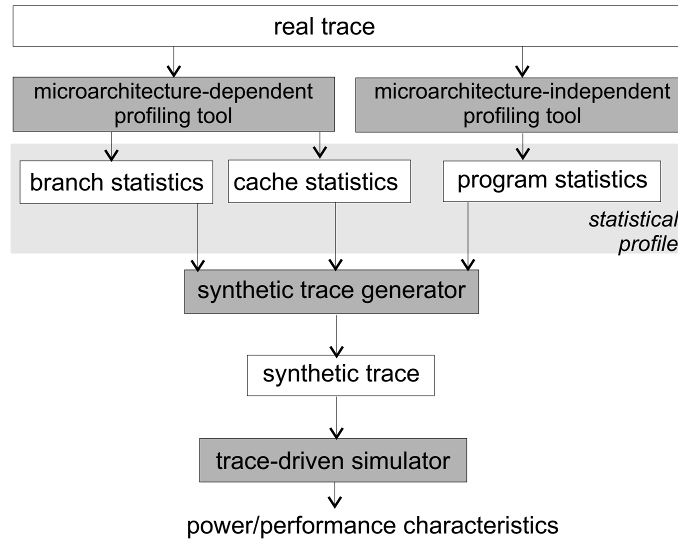


Figure 2.2: Statistical simulation: general framework.

statistical simulation is useful for embedded system designs (typically small-resource machines due to power considerations) as well as for high-performance designs (typically wide-resource machines for optimal performance).

In the evaluation section of this chapter, we will present experiments in which we assume (partially) idealized microarchitectures in order to validate the accuracy of the various components in the statistical simulation method. We will for example assume that the branch prediction is perfect which means that every branch is correctly predicted by the dynamic branch predictor, i.e., no branch mispredictions occur. In other experiments, we assume perfect caches, or every access to the cache is a hit.

2.2 Statistical simulation

The general framework of statistical simulation is depicted in Figure 2.2 and consists of three steps: *statistical profiling*, *synthetic trace generation* and *trace-driven simulation*. This framework closely resembles the framework initially proposed by Carl and Smith [20] and has been evaluated concurrently by a number of research groups over the recent

years [37, 39, 102, 106]. This related work will be extensively discussed at the end of chapter 3.

In the first step, a program trace—e.g., a SPEC benchmark trace—is analyzed by a *microarchitecture-dependent* profiling tool and a *microarchitecture-independent* profiling tool. The microarchitecture-independent profiling tool extracts statistics concerning program characteristics: a well chosen set of distributions is measured concerning the instruction mix and the dependencies between instructions through register values as well as through memory values. The microarchitecture-dependent profiling tool extracts statistics concerning the branch and cache behavior of the program trace for a specific branch predictor and a specific cache organization. This is done by simulating the desired aspect, namely the branch or the cache behavior. Another way of computing these statistics could be by means of an analytical model that estimates the branch and cache behavior for the application and a given branch predictor and cache organization; however, using estimated characteristics will lead to additional inaccuracies when generating synthetic traces. The complete set of statistics (program, branch and cache statistics) computed through statistical profiling, is called a *statistical profile*. Note that a statistical profile needs to be computed only once for each benchmark.

Once a statistical profile is computed, a synthetic trace generator generates a *synthetic trace* using this statistical profile (second step). The generated synthetic trace will have the same execution properties, by construction, as the original trace from which the statistical profile was derived to construct the synthetic trace. This, of course, only holds for the characteristics included in the statistical profile; other execution characteristics that are not included in the statistical profile might not be modeled adequately, e.g., instruction throughput as a function of time.

The synthetic trace can now be simulated on a trace-driven simulator (third step). If the synthetic trace captures the right execution characteristics, the evaluation characteristics, e.g., the number of instructions retired per clock cycle (IPC), of the original trace and the synthetic trace should be comparable when simulating the same architecture. While simulating this synthetic trace, performance characteristics typically convergence after a few millions of instructions due to the statistical nature of the technique, see section 2.8.

Discussion. Ideally, we would only include microarchitecture-independent characteristics in our statistical profile. Indeed, we would like to use statistical simulation for microprocessor design space explorations while varying a broad range of microarchitectural parameters, such as the number of functional units, the instruction window size, the cache size, the branch predictor, etc. In practice however, program characteristics dealing with locality, such as cache and branch behavior, are hard to model accurately using microarchitecture-independent characteristics. Therefore, the statistical simulation methodology as discussed in this dissertation makes a distinction between microarchitecture-dependent (only concerning program locality properties) and microarchitecture-independent characteristics. As a result, a major and an important part of the design space can still be explored through statistical simulation using a single statistical profile: e.g., the instruction window size, the number of functional units, the instruction execution latency, the number of pipeline stages, etc. can be varied freely. However, separate statistical profiles need to be computed for various cache configurations and branch predictors. Note that this is not a major drawback since there exist fast methods for simulating multiple cache configurations simultaneously, for example cheetah [128].

Note also that a statistical profile can be computed from an actual trace that is stored on a disk, but it is more convenient to compute it on-the-fly from either an instrumented functional simulator, or from an instrumented version of the benchmark program running on a real system. Indeed, storing traces on a disk can be impractical or even impossible when representative traces are needed from long running workloads.

A final note is that a statistical profile is not only dependent on the application. It is also a function of the input to the application, the compiler and the instruction set architecture (ISA). However, this does not affect the applicability of statistical simulation because this method is intended to be used during the definition of the microarchitecture. At that point in the design process, the workload being used and the ISA are fixed already.

2.3 Applications

The statistical simulation methodology has several interesting applications:

- **Performance evaluation.** The most obvious application is efficient performance evaluation. Since statistical simulation is quite accurate and fast, performance estimations are obtained efficiently. As such, the design space can be explored in an early design stage. The aim of statistical simulation is not to replace detailed cycle-accurate simulations, but to identify an interesting area in the design space that can be further analyzed through more detailed simulations. Since the design space can be explored very fast through statistical simulation and since the area of interest to be further analyzed is limited in size, we can conclude that statistical simulation will succeed in its goal, namely to reduce the overall simulation time. In this chapter and the next chapter, a number of examples will be given that illustrate the use of statistical simulation in performance evaluation.

  Hennessy and Patterson [64] present the following misbelief or fallacy in their reference book on computer architecture: *synthetic benchmarks predict performance for real programs*. In their discussion on this fallacy, they allude on Whetstone by Curnow and Wichmann [26] in the 1970s and Dhrystone by Weicker [136] in the 1980s. These two synthetic programs[4] do not reflect real program behavior of today. The synthetic traces generated in this dissertation could be viewed as a second generation of synthetic benchmark generation that is useful for current microprocessor designs.

- **System evaluation.** For larger systems containing several processors, such as multiprocessors, clusters of computers, etc., simulation time is even a bigger problem since all the individual components in the system need to be simulated simultaneously. An interesting example is given in [103] in which Nussbaum and Smith evaluate symmetric multiprocessor systems through statistical simulation.

- **Workload characterization.** Another interesting application for statistical simulation is workload characterization. While validating the statistical simulation methodology in general and the characteristics included in the statistical profile in particular, it will become clear what program characteristics need to be included in the profile in order to obtain a higher accuracy. As

[4]Unfortunately, Dhrystone is still being used in the emerging embedded market.

such, program characteristics that have an influence on performance will be discriminated from characteristics that do not influence performance. Examples of this application will be given in chapters 3 and 4.

- **Workload space exploration.** Since a statistical profile basically consists of a number of program characteristics, these characteristics can be easily varied and the influence of these parameters on overall performance can be measured. Note that these characteristics can be varied independently from the other characteristics. If desired, several characteristics can be varied simultaneously. In short, statistical simulation allows researchers to explore the workload space whereas a benchmark only presents one single point in this huge workload space. An example of workload space exploration is given chapter 4.

- **Power modeling.** As discussed in the introduction of this dissertation, power consumption becomes a key design issue when designing contemporary microprocessors. As such, power consumption should be incorporated in early stages of the design cycle so that computer designers are not confronted with an unexpected power consumption in a late design stage. By incorporating an architectural power model in the trace-driven simulator for simulating synthetic traces, power consumption can be estimated through statistical simulation in the earliest stages of design process. Chapter 5 presents and evaluates statistical simulation for power modeling.

2.4 Statistical profiling

True wisdom is to know what to leave out.
Advice from David Patterson to
Bill Joy, see foreword of [64]

While searching for a viable statistical profile, three major goals need to be fulfilled. First, the performance characteristics of the synthetic trace on a particular architecture should be comparable to the performance characteristics of the corresponding original trace. In this

dissertation, we focus on modeling out-of-order processor performance for which accurate modeling of program parallelism is critical to obtain accurate performance predictions. Therefore, a viable statistical profile should be selected carefully.

A second important aspect is that the resulting statistical profile should not be too complicated, i.e., the number of distributions involved should be limited while preserving high levels of performance prediction accuracy. It is obvious that by incorporating more distributions in a statistical profile, the resulting synthetic trace will resemble more the original trace, which should lead to more accurate performance predictions. But the number of probabilities to be stored in a statistical profile might explode when including too many distributions, which would make this technique impractical.

A third goal is to generate benchmark traces that are syntactically correct. More specifically, a store operation should not have a destination operand and an integer operation should not have four or five source operands. The underlying motivation for this goal is that a synthetic trace should resemble a real trace as much as possible. In addition, the syntactical correctness of synthetic traces allows us to simulate synthetic traces on existing trace-driven simulators (with minor modifications needed to the simulation software). The goal of syntactical correctness can be fulfilled by both carefully selecting a statistical profile and by carefully designing the synthetic trace generator, as will become clear in the following sections.

In the next two subsections, we discuss the microarchitecture-independent and the microarchitecture-dependent characteristics included in our statistical profile.

2.4.1 Microarchitecture-independent characteristics

The first microarchitecture-independent program characteristic is the *instruction mix* distribution. We measure the probability that the type T_x of instruction x equals t, for t one of the N_{instr} instruction types included in the model. Formally stated, we measure $P\,[\,T_x = t\,]$. In our model which is set up for the Alpha ISA, we identify 13 instruction classes ($N_{instr} = 13$) because of their varying execution latencies and semantics: integer arithmetic, load, store, conditional branch, unconditional jump with offset, indirect jump, call with offset, indirect call, return, integer multiply, floating-point operation, floating-point divide

single-precision and floating-point divide double-precision.[5]

The second distribution measured is the *number of operands* distribution or the probability that instruction x has O_x source operands. Since the number of operands of an instruction is dependent on its type, we actually measure $P\,[\,O_x = o\,|\,T_x = t\,]$. The variable number of operands is caused by the fact that some instructions occur in a register-register as well as in a register-immediate format, while belonging to the same instruction class t in our classification.

To model dependencies through register values, we also measure the *age of register operands* distribution for each register operand of each instruction type. We measure the probability that the i-th register operand of an instruction x of type T_x having O_x operands, is produced δ instructions before it in the trace; i.e., instruction $x - \delta$ produces a register instance (register write) that is consumed by instruction x (register read). Formally stated, we measure $P\;[\;A_{i,x} = \delta \mid T_x = t, O_x = o\,]$, with $i = 1, 2$ and $A_{i,x}$ the age of the i-th register operand of instruction x. Notice that in these distributions only read-after-write (RAW) data dependencies are considered. Write-after-write (WAW) and write-after-read (WAR) dependencies are not included because the evaluation in this dissertation is aimed at out-of-order architectures in which WAW and WAR dependencies are removed dynamically through register renaming.[6]

To include memory dependencies in our synthetic traces, we also measure the probability that a load operation accesses the same memory address as a store operation that occurs ahead in the instruction stream. We identify the probability that a load is *memory-dependent* on the δ-th store before it in the trace (read-after-write dependency through memory). In other words, we measured the age of memory instances distribution.

Note that the age of register operands distribution and the age of memory operands distribution are theoretically infinite. But for practical purposes, they can be truncated at a certain limit N_{max}. This limit imposes a constraint on the window size of the out-of-order architectures being modeled, since inter-operation communication is not mod-

[5]In earlier days of computing, the instruction mix combined with the execution time of each instruction type, or the average instruction execution time, gave an accurate performance estimate. A popular example in those days was the Gibson mix: transfers to and from memory (31%), indexing (18%), branching (17%), floating-point arithmetic (12%), fixed-point arithmetic (7%), shifting (4%) and miscellaneous (11%).

[6]Note that this assumes perfect register renaming.

eled for communication distances larger than N_{max}. In this dissertation, we choose $N_{max} = 512$ which allows the exploration of all contemporary and near future out-of-order architectures.

2.4.2 Microarchitecture-dependent characteristics

The microarchitecture-dependent characteristics consist of the branch statistics and the cache statistics.

Branch statistics

The branch statistics consist of seven probabilities: (i) the conditional branch target prediction accuracy, (ii) the conditional branch (taken/not-taken) prediction accuracy, (iii) the (unconditional) relative branch (with offset) target prediction accuracy, (iv) the relative call (with offset) target prediction accuracy, (v) the indirect jump target prediction accuracy, (vi) the indirect call target prediction accuracy and (vii) the return target prediction accuracy. The reason to distinguish between these seven probabilities is that the prediction accuracies greatly vary among the various branch classes. In addition, the penalties introduced by these are completely different [64]. For example, in case of a conditional branch, a target *mis*prediction together with a correct taken/not-taken prediction results in a single-cycle bubble in the pipeline; a target *mis*prediction combined with an *in*correct taken/not-taken prediction will require the entire processor pipeline to be flushed when the mispredicted branch is executed. A misprediction in cases (iii) and (iv) only introduces a single-cycle bubble in the pipeline. Cases (v), (vi) and (vii) on the other hand, will cause the entire processor pipeline to be flushed.

Cache statistics

The cache statistics include two sets of distributions: the data cache statistics and the instruction cache statistics. The data cache statistics contain two probabilities, namely the probability that a load operation needs to access the L2 cache (in case of a L1 D-cache miss) and main memory (in case of a L2 cache miss) to get its data, respectively. The instruction cache statistics consist of two probabilities as well, namely the probability that the fetch unit needs to access the L2 cache and main memory to get an instruction, respectively.

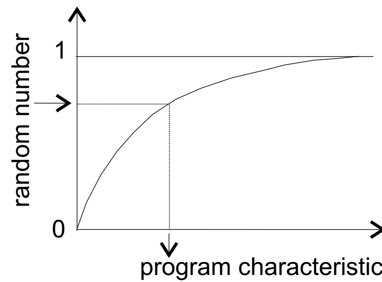


Figure 2.3: Determining program characteristics through random number generation.

2.5 Synthetic trace generation and simulation

Once a statistical profile is computed, a *synthetic trace* can be generated by a synthetic trace generator using this statistical profile.

This is based on a Monte Carlo method, see Figure 2.3: a random number is generated between 0 and 1 which will be uniformly distributed along this interval. Using the cumulative distribution function, a program characteristic is determined. By construction, this program characteristic will then be distributed as specified by the cumulative distribution function.

The generation of a synthetic trace itself works on an instruction-by-instruction basis: the first instruction has number 0, the second has number 1, etc. Consider the generation of instruction x in the synthetic instruction stream, see Figure 2.4:

1. Determine the instruction type and the number of source operands using the instruction mix distribution and the number of operands distribution; e.g., an add, a store, a branch, etc. were generated in Figure 2.4.

2. For each source operand, determine the instruction that creates this register instance using the age of register instances distribution. Notice that when a dependency is created in this step, the demand for syntactical correctness does not allow us to assign a destination operand to a store, a branch, a jump or a return instruction. For example in Figure 2.4, the load instruction cannot be made dependent on the preceding branch. However, using the Monte Carlo method we cannot assure that the instruction that is

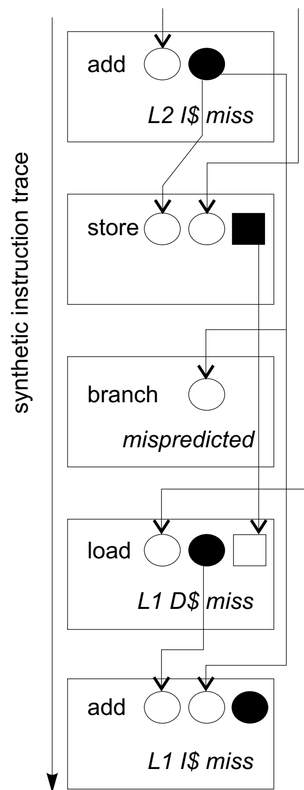


Figure 2.4: Synthetic trace generation.

the creator of that register instance, is neither a store, a branch, a jump nor a return instruction. This problem is solved as follows: look for another creator instruction until we get one that is not a store nor a conditional branch. If after a certain maximum number of trials, in our experiments after 10,000 trials, still no dependency is found that is not supposedly created by a store, a branch, a jump or a return, the dependency is simply removed. At the end of this section, we detail on the implications of this approach.

3. If instruction x is a load instruction, use the age of memory instances distribution to determine whether a store instruction with number w (before instruction x in the trace; i.e., $w < x$) accesses the same memory address; e.g., a read-after-write dependency is imposed through memory between the load and the store in Figure 2.4. This will have its repercussions when simulating these instructions. In our simulator we assume speculative out-of-order execution of memory operations. This means that when a load x that accesses the same memory location as a previous store w ($w < x$), is executed earlier than the store, the load would get the wrong data. To prevent this, a table is kept in the processor, e.g., an Address Resolution Buffer (ARB) [56], to keep track of memory dependencies. When the store w is executed later, it will detect in that table that load x has accessed the same memory location. In that case, the load and all its dependent instructions need to be re-executed.

4. If instruction x is a branch, determine whether the branch and its target will be correctly predicted using the branch statistics. The appropriate penalty, i.e., a single-cycle bubble or a pipeline flush, will then be imposed at the right time during simulation: a bubble is introduced in the pipeline in the fetch stage or the pipeline is flushed when the mispredicted branch is executed. In order to model resource contention due to branch mispredictions, we take the following action while simulating a synthetically generated trace: when a 'mispredicted'-labeled branch is inserted in the processor pipeline, instructions are injected in the pipeline (also synthetically generated) to model the fetching from a misspeculated control flow path. These instructions are then marked as coming from a misspeculated path. When the misspeculated branch is executed, the instructions on the misspeculated path are removed,

new instructions are fetched (again synthetically generated) and marked as coming from the correct control flow path.

5. If instruction x is a load instruction, determine whether the load will cause a L1 cache hit/miss or L2 cache hit/miss using the data cache statistics. When a 'L1 cache miss'-labeled load instruction or a 'L2 cache miss'-labeled load instruction is executed in the pipeline, the simulator assigns an execution latency according to the type of the cache miss. In case of a L1 cache miss, the L2 cache access time will be assigned; in case of a L2 cache miss, the memory access time will be assigned, see Table 2.1 for the actual access times.

6. Determine whether or not instruction x will cause an instruction cache hit/miss at the L1 or L2 level. In Figure 2.4, the first and the last instruction get the label 'L2 cache miss' and 'L1 cache miss', respectively. When a 'L1 cache miss'-labeled instruction or a 'L2 cache miss'-labeled instruction is inserted into the pipeline, the processor will stop inserting new instructions in the pipeline during a number of cycles. This number of cycles is the L2 cache access time or the memory access time in case of L1 cache miss or a L2 cache miss, respectively, see Table 2.1 for the actual access times.

The last phase of the statistical simulation method is the trace-driven simulation of the synthetic trace which yields estimates of performance characteristics. An important performance characteristic is the average number of instructions executed per cycle (IPC) which can be easily calculated by dividing the number of instructions simulated by the number of simulation cycles.

The approach for guaranteeing syntactical correctness as described under the second bullet of the synthetic trace generation algorithm has a serious implication. In Figure 2.5, the deviation is shown between the desired (marginal) age of register operands distribution—measured from the real program trace—and the (marginal) age of register operands distribution of the generated synthetic trace. This graph was obtained by subtracting the distribution of the synthetic trace from the distribution of the real trace. We observe that the distribution resulting from synthetic trace generation (generally) lies under the distribution of the real trace[7]. This is due to the fact that if no de-

[7]Note that this is only true for small dependency distances; for larger dependency

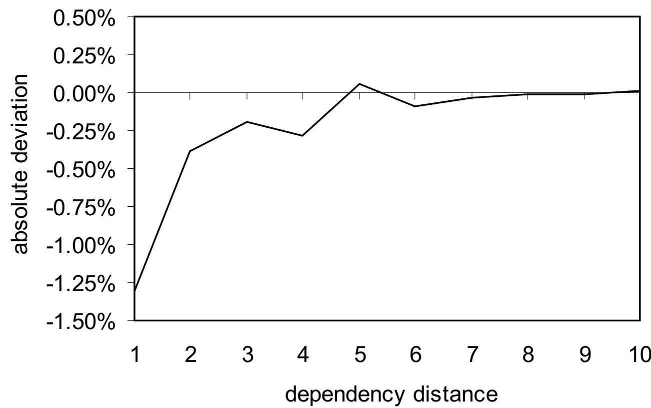


Figure 2.5: The deviation between the marginal age-of-register-operands distribution of the original and the synthetic trace (for the li benchmark) as a function of the dependency distance. A deviation of -1% means that for example the probability of the original trace is 16% whereas the probability of the synthetic trace is 15%.

pendency is found that is not supposedly created by a store, a branch, a jump or a return, the instruction is made dependent on an instruction that comes at least $N_{max} = 512$ instructions before it in the trace which is basically the same as removing the dependency.

From these considerations we can conclude that it cannot be guaranteed that the statistical profile of the synthetic trace equals the statistical profile of the real program trace. Therefore, we use *instantaneous positive-error* distributions, see Figure 2.6 for an example. An instantaneous positive-error distribution is attained by computing the errors between the desired probabilities and the probabilities at that time during the synthetic trace generation process, the *instantaneous* distribution, only keeping the positive errors and normalizing them to a distribution. When dependencies are generated using the instantaneous positive-error distribution, dependency distances whose instantaneous probability is lower than the desired probability, will be benefited. At the same time, dependency distances whose instantaneous probability is higher than the desired probability, will be harmed. In fact, the use of

distances, in our case more than N_{max}, the opposite will be true since distributions sum to one.

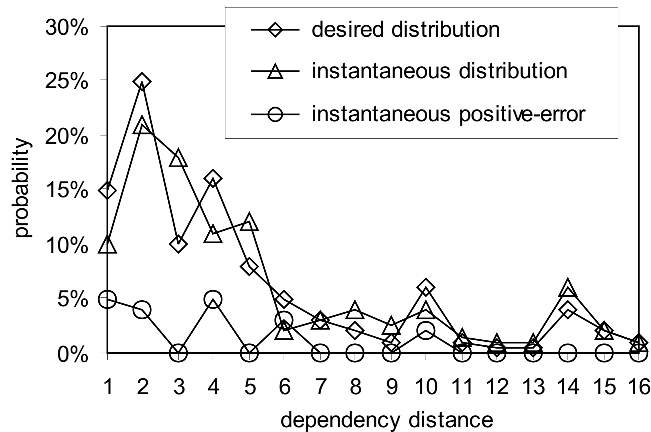


Figure 2.6: Example illustrating the feedback loop implemented in the synthetic trace generator: the desired distribution, the instantaneous distribution and the instantaneous positive error.

the instantaneous positive-error distribution introduces a *feedback loop* in the synthetic trace generation algorithm. As a result, the synthetic trace will have the same statistical profile as the original trace, which was verified. Through experimentation, we verified that implementing this feedback loop slightly increases the performance prediction accuracy.

2.6 Methodology

The benchmark traces used in this study are from the SPECint95 suite[8] and the IBS suite [131], see Tables 2.2, 2.3 and 2.4. The SPECint95 traces were generated on a DEC 500au station with an Alpha 21164 processor. The Alpha architecture is a load/store architecture and has 32 integer and 32 floating-point registers, each of which is 64 bits wide. The SPECint95 benchmarks have been compiled with the DEC cc compiler version 5.6 with the optimization flag set to -O4 and linked statically using the -non_shared flag. The traces were carefully selected not to include initialization code. The IBS traces were generated on a MIPS-based DEC 3100 system running the Mach 3.0 operating system. These

[8] `http://www.spec.org`

SPECint95		
benchmark	description	input
li	Xlisp interpreter	train.lsp
go	Go-playing game	50 9 2stone9.in
compress	Text compressing	< train.in (50K)
gcc	GNU C compiler 2.5.3	gcc.i -O -funroll-loops -finline-functions
m88ksim	Motorola 88100 simulator	-c < train.in
ijpeg	Image (de)compression	penguin.ppm -subimage 0 0 0.5 0.5
perl	Perl interpreter	scrabbl.pl < scrabbl.in
vortex	Object-oriented database	vortex.in (persons.250)
IBS		
benchmark	description	
mpeg	mpeg_play v2.0 displays 85 frames from compressed video	
jpeg	xloadimage v3.0 displays two JPEG images	
gs	Ghostscript v2.4.1 displays page of text and graphics in an X window	
verilog	Verilog-XL v1.6b simulating logic design of microprocessor	
real_gcc	GNU C compiler v2.6	
sdet	Multiprocess performance benchmark from the SPEC SDM suite	
nroff	Unix text formatting shipped with Ultrix 3.1	
groff	GNU C++ implementation v1.09 of Unix nroff	

Table 2.2: The benchmarks used for the evaluation: SPECint95 and IBS.

traces contain significant amounts of operating system activity, 38% on average. We included these traces in our study because these traces are known to stress the memory subsystem more than SPECint benchmarks do [131]. The traces used are relatively small. However, we believe that this does not impact the evaluation of our technique since a large trace can be split up into smaller traces if using one statistical profile for a long trace would ever prohibit accurate performance modeling.

2.7 Performance prediction accuracy

In the first part of this evaluation section, we discuss the absolute accuracy of statistical simulation. Afterwards, we will detail on the relative accuracy.

	li	gcc	compress	go	ijpeg	vortex	m88ksim	perl
dyn. instr. count (M)	226	182	217	200	170	200	200	200
stat. instr. count (90%)	580	24,084	336	6,744	1,150	2,544	1,097	1,546
% conditional branches	11.22%	12.70%	7.17%	10.10%	8.69%	9.33%	7.67%	10.95%
prediction accuracy	93.99%	91.49%	89.17%	79.01%	90.91%	99.03%	95.87%	96.21%
% loads	34.36%	30.18%	31.02%	37.19%	24.70%	28.38%	29.12%	29.84%
% stores	13.83%	10.98%	4.80%	7.32%	5.21%	14.62%	14.09%	14.60%
8KB DM L1 I-cache; 8KB DM L1 D-cache; 64KB 2WSA unified L2 cache								
L1 I-cache miss rate	2.17%	3.87%	<0.01%	3.53%	0.03%	5.68%	3.67%	4.30%
L2 I-cache miss rate	0.03%	0.95%	<0.01%	0.65%	<0.01%	0.50%	1.95%	0.38%
L1 D-cache miss rate	4.42%	7.10%	3.88%	8.32%	3.42%	10.63%	6.66%	16.50%
L2 D-cache miss rate	2.67%	2.27%	2.42%	0.73%	0.39%	1.33%	0.25%	1.43%
32KB DM L1 I-cache; 64KB 2WSA L1 D-cache; 256KB 4WSA unified L2 cache								
L1 I-cache miss rate	0.03%	1.67%	<0.01%	1.38%	<0.01%	1.59%	3.85%	1.02%
L2 I-cache miss rate	<0.01%	0.19%	<0.01%	0.01%	<0.01%	0.03%	<0.01%	<0.01%
L1 D-cache miss rate	2.43%	1.12%	1.45%	0.06%	0.02%	0.27%	0.01%	0.54%
L2 D-cache miss rate	0.20%	0.28%	0.97%	0.03%	0.36%	0.46%	0.02%	0.02%

Table 2.3: More details on the SPECint95 traces.

	mpeg	jpeg	gs	verilog	real_gcc	sdet	nroff	groff
dyn. instr. count (M)	99	97	106	47	110	38	110	97
stat. instr. count (90%)	7,357	1,740	7,895	5,631	21,940	7,804	2,014	4,346
% conditional branches	9.63%	16.31%	14.45%	13.68%	15.01%	10.68%	20.81%	12.89%
prediction accuracy	95.27%	98.73%	97.15%	97.04%	93.82%	95.35%	98.23%	97.24%
% loads	25.03%	16.93%	22.66%	27.74%	24.26%	24.81%	21.54%	26.63%
% stores	17.77%	7.42%	14.11%	20.35%	13.92%	16.38%	10.23%	15.00%
8KB DM L1 I-cache; 8KB DM L1 D-cache; 64KB 2WSA unified L2 cache								
L1 I-cache miss rate	2.40%	1.42%	4.09%	3.99%	3.57%	3.62%	3.06%	5.41%
L2 I-cache miss rate	1.87%	0.96%	1.18%	1.11%	1.18%	2.45%	0.40%	1.13%
L1 D-cache miss rate	7.33%	4.26%	6.72%	4.55%	5.33%	5.56%	6.82%	4.85%
L2 D-cache miss rate	4.67%	3.01%	2.58%	7.94%	2.39%	4.75%	1.15%	1.79%
32KB DM L1 I-cache; 64KB 2WSA L1 D-cache; 256KB 4WSA unified L2 cache								
L1 I-cache miss rate	1.86%	1.22%	2.14%	1.77%	1.94%	2.98%	1.96%	2.70%
L2 I-cache miss rate	0.39%	0.10%	0.19%	0.24%	0.28%	0.45%	0.05%	0.09%
L1 D-cache miss rate	1.84%	0.93%	0.59%	3.62%	0.70%	1.32%	0.48%	0.46%
L2 D-cache miss rate	1.17%	0.72%	0.83%	3.54%	0.62%	1.61%	0.23%	0.37%

Table 2.4: More details on the IBS traces.

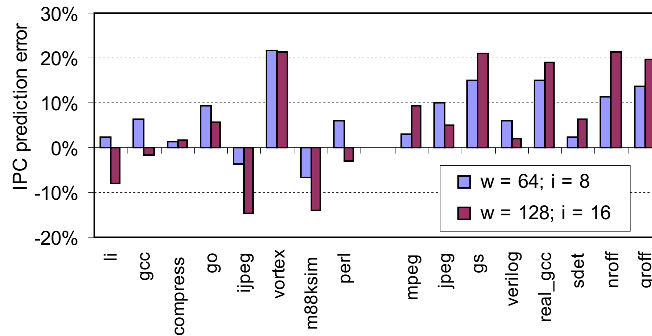


Figure 2.7: Modeling instruction-level parallelism: instruction mix and inter-operation dependencies.

2.7.1 Absolute accuracy

The metric used to measure the absolute accuracy is the *IPC prediction error* which is defined as follows:

$$\textit{IPC prediction error} = \frac{\textit{IPC synthetic trace} - \textit{IPC real trace}}{\textit{IPC real trace}}. \qquad (2.1)$$

In other words, the IPC prediction error measures the inaccuracy in IPC when comparing the real trace and a synthetic trace generated using the statistical profile corresponding to the real trace. Note that a smaller IPC prediction error is better (higher accuracy) and that a positive vs. negative prediction error implies an IPC overestimation vs. underestimation, respectively. In this section, we will also use the *average prediction error* which is defined as the arithmetic average over the *absolute values* of the prediction errors for the individual benchmarks.

Before proceeding to the overall performance prediction accuracy, we first evaluate the accuracy of the individual components: modeling instruction-level parallelism (ILP), branch behavior, I-cache behavior and D-cache behavior. This will help us in understanding the overall prediction errors.

Instruction-level parallelism

In this paragraph, we evaluate the capability of the statistical simulation methodology for modeling instruction-level parallelism, i.e., for

modeling the instruction mix and inter-operation dependencies. For this purpose, we have set up the following experiment. We have simulated a real trace while assuming perfect branch prediction and perfect caches—every branch is correctly predicted and every access to the cache is a hit. This gives us the number of instructions retired per clock cycle (IPC) for the real trace. The IPC of the synthetic trace was obtained by simulating a synthetically generated trace under the same assumptions (perfect branch prediction and perfect caches). In other words, we only evaluate the instruction mix distribution, the number of operands distribution, the age of register operands distribution and the age of memory instances distribution; i.e., the microarchitectural-independent characteristics as discussed in section 2.4.1, or steps 1, 2 and 3 of the synthetic trace generator algorithm, see section 2.5. The results of this experiment are shown in Figure 2.7 for an 8-wide, 64-entry window processor and a 16-wide, 128-entry window processor. The prediction errors vary between -14.6% and 21.7%. The average prediction error for the 8-wide, 64-entry window machine is 8.6% and 8.7% for the SPECint95 traces and the IBS traces, respectively. The average prediction error for the 16-wide 128-entry window configuration is higher: 11.4% and 16.2% for the SPECint95 traces and the IBS traces, respectively. The fact that the error is higher for a wider processor configuration implies that modeling the performance of wide-resource machines is a greater challenge than modeling the performance of small-resource machines. This can be explained intuitively by the fact that performance is more limited by program parallelism than by machine parallelism for wider machines.

In the next chapter, we will discuss how to better model the ILP.

Branch behavior

This paragraph evaluates statistically modeling branch (mis)prediction behavior. The IPC of the real trace was obtained by simulating the real trace while assuming perfect caches and a realistic branch predictor, see Table 2.1 for the exact configuration of the branch predictor. The IPC of the synthetic trace was obtained by simulating (under the same assumptions) a *real* trace annotated with statistically generated branch (mis)prediction information, i.e., step 4 of the synthetic trace generation, see section 2.5. As such, instead of accessing the branch predictor when executing a branch instruction, the annotation is used to incur an appropriate penalty in the simulator. The results of this experi-

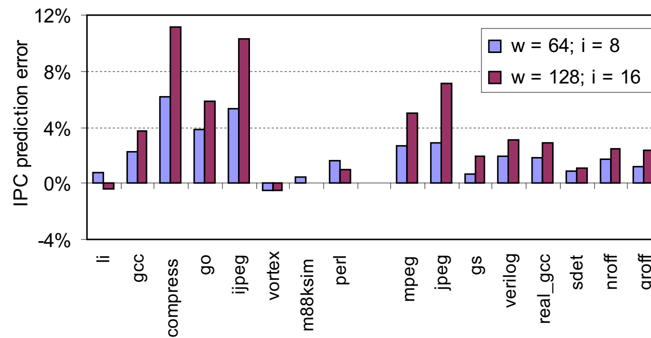


Figure 2.8: Modeling branch behavior.

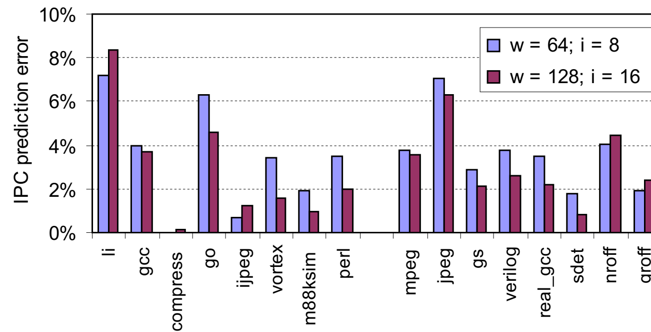


Figure 2.9: Modeling I-cache behavior: case of 'small' cache configuration.

ment are shown in Figure 2.8. The prediction errors vary between -0.5% and 11.2%. The average prediction error for the 64/8 machine is 2.6% and 1.7% for the SPECint95 traces and the IBS traces, respectively; for 128/16 configuration, the average prediction error is 4.1% and 3.2%, respectively. We can conclude that the branch behavior is modeled quite accurately.

I-cache behavior

In this paragraph, we evaluate the accuracy of modeling the I-cache behavior by means of two parameters, namely the L1 I-cache miss rate and the L2 cache miss rate as discussed previously in section 2.4.2. The

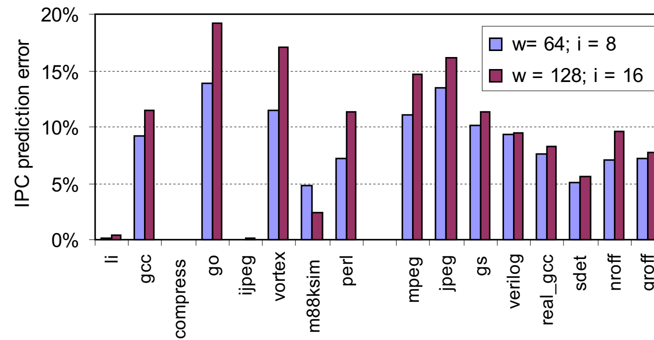


Figure 2.10: Modeling I-cache behavior: case of 'large' cache configuration.

IPC of the real trace is obtained by simulating the real trace on a microarchitecture with a perfect branch predictor and a perfect D-cache, i.e., every load operation results in a cache hit. The IPC of the synthetic trace is obtained by simulating (under the same assumptions) a *real trace* annotated with statistically generated I-cache miss information, step 6 of the synthetic trace generation algorithm, see section 2.5. The results of this experiment are shown in Figures 2.9 and 2.10 for the 'small' and the 'large' cache configuration (see Table 2.1), respectively.

In all cases the prediction errors are positive; in other words, IPC is overestimated. This can be explained as follows: the synthetic generation of I-cache miss information as discussed so far will result in I-cache misses that are uniformly distributed over the trace. In reality however, cache miss behavior is bursty [129, 134]. Not modeling this bursty behavior certainly has its implications on the performance prediction accuracy. Indeed, the impact on performance of a burst of cache misses is different than the impact (of the same amount) of cache misses that are uniformly distributed since the state of the processor will be significantly different in both situations. In the first case, the instruction window might be sparsely filled when a cache miss occurs shortly after another cache miss; in the other case, the instruction window might be (nearly) completely filled with useful instructions so that useful work can still be done while waiting for the cache miss to resolve. This will hide the memory latency to some extent. As a result, modeling cache behavior using cache statistics that yield a flat or uniform cache miss behavior, leads to non-negligible IPC overestimations. In the next chapter,

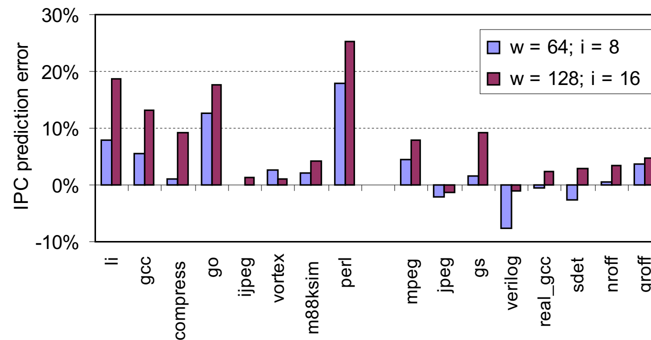


Figure 2.11: Modeling D-cache behavior: case of 'small' cache configuration.

we will show how this bursty behavior can be modeled and how this leads to smaller prediction errors.

The prediction errors in Figures 2.9 and 2.10 are larger for the 'large' cache configuration than for the 'small' cache configuration: up to 19.3% versus 8.4%, respectively. The reason for this is again the bursty behavior of I-cache misses. For the 'small' cache configuration, the cache misses are more uniformly distributed than for the 'large' cache configuration since there are more cache misses. As such, the cache miss behavior more closely resembles the behavior generated by our method, leading to smaller prediction errors.

Note that for some benchmarks and some cache configurations, the prediction errors are nearly zero. For example, in case of the 'large' cache configuration, the prediction errors are close to zero for li, compress and ijpeg. This is due to the extremely low I-cache miss rates, see Table 2.3.

D-cache behavior

To evaluate the accuracy of modeling the D-cache behavior by means of two probabilities, namely the L1 and L2 D-cache miss rates, we have set up an experiment similar to the experiments from the previous paragraphs. The IPC of the real trace was obtained by simulating the real trace while assuming perfect branch prediction and a perfect I-cache. The IPC of the synthetic trace was obtained by simulating (under the same assumptions) a real trace annotated with statistically generated

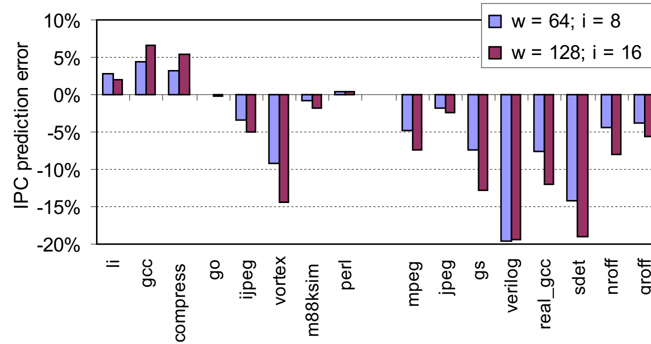


Figure 2.12: Modeling D-cache behavior: case of 'large' cache configuration.

D-cache miss information, step 5 of the synthetic trace generation algorithm, see section 2.5. The results of this experiment are shown in Figures 2.11 and 2.12 for the 'small' and the 'large' cache configuration (see Table 2.1), respectively. The prediction errors vary between -19.5% and 25.2% which is quite high. There are several explanations for this:

- we do not model *delayed hits*. A delayed hit happens for example, when two loads access the same cache block and when the first load is a cache miss, then the second load is a delayed hit. Indeed, the second load will see a long latency while the first load is being resolved; however, the second load is classified as a cache hit (since it is a cache hit when simulating instruction per instruction). As such, our simulator will assign a cache hit latency to the second load which is an optimistic approximation.

- we do not model store misses either. Store misses can have an impact on performance through delayed (load) hits as well.

- in addition, we do not model the bursty D-cache miss behavior.

- and finally, we do not take into account the latency tolerance of load operations [124]. Latency tolerance means that for some load operations the execution latency does influence the overall performance while for other load operations it does not. This is dependent on the dependency graph, i.e., on the number of instructions and their types that are (directly or indirectly) dependent on that load.

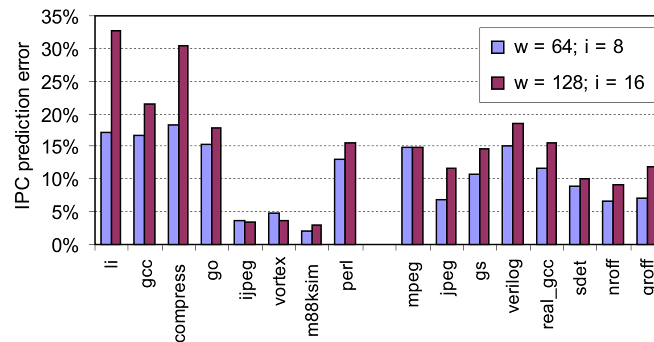


Figure 2.13: Overall prediction accuracy: case of 'small' cache configuration.

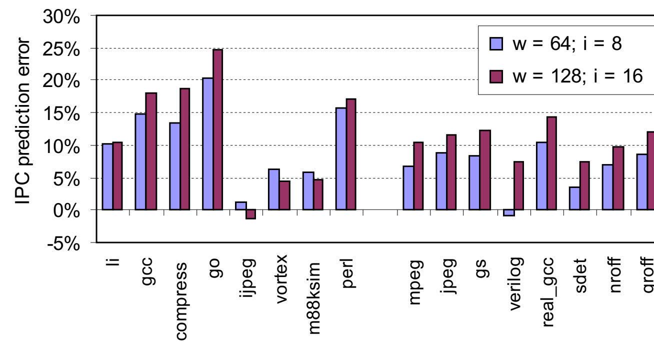


Figure 2.14: Overall prediction accuracy: case of 'large' cache configuration.

In the next chapter, we will discuss how to address some of these short-comings.

Overall prediction accuracy

Finally, we can evaluate the overall performance prediction accuracy, i.e., by comparing the IPC of a real trace with the IPC of a synthetic trace. The IPC of the real trace is obtained by simulating the trace on a microarchitecture with a real branch predictor and real caches. The IPC of the synthetic trace is obtained by generating and simulating a synthetic trace using the complete statistical profile as discussed in section 2.5. The results are shown in Figures 2.13 and 2.14 for the 'small'

and the 'large' cache configuration, respectively. The prediction errors vary between -1.3% and 32.8% and are larger for a processor with more resources: e.g., for the 'large' cache configuration and for the IBS traces, the average error is 6.8% for the 8-wide 64-entry window processor versus 10.7% for the 16-wide 128-entry window processor. The largest average error is observed for the SPECint95 traces on a 16-wide 128-entry window processor with the 'small' cache configuration, namely 16.0%.

Some of the prediction errors observed in Figures 2.13 and 2.14 can be explained in terms of the individual components, namely ILP, branch behavior, I-cache behavior and D-cache behavior. For example, for the 'small' cache configuration and perl, the prediction error is about 15%. The errors for the individual components for the ILP, the branch behavior and the I-cache behavior are no larger than a few percent. The error for the D-cache behavior on the other hand is quite large, i.e., more than 20%. As such we can conclude that the overall prediction error basically comes from not modeling the D-cache behavior adequately.

Other overall prediction errors are harder to be explained in terms of the individual components. As such these errors have to be explained by the notion of interaction between the various program characteristics. For example, the prediction error of the 'large' cache configuration for li is about 10% for the 64/8 and the 128/16 configuration. On the other hand, Figure 2.7 shows a 2.5% and a -8.3% prediction error concerning the ILP for the 64/8 and the 128/16 configuration, respectively. Figures 2.8 and 2.10 show a prediction error close to zero for the branch behavior and the I-cache behavior, respectively. Figure 2.12 shows a 2.7% and a 1.9% prediction error concerning the D-cache for the 64/8 and the 128/16 processor, respectively. As such, the overall prediction error cannot be explained in terms of the individual components—ILP, branch behavior, I- and D-cache behavior—which leads to the conclusion that an additional source causes these overall prediction errors, namely the interaction between the various characteristics.

To better understand this interaction we have set up the following experiment. In Figure 2.15, we compare the IPC prediction errors in four situations:

1. when statistically modeling the ILP, the branch prediction behavior and the cache behavior;

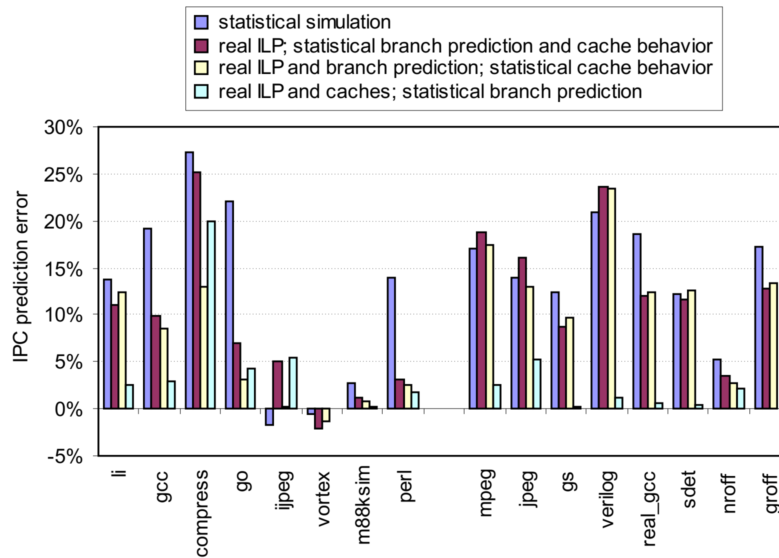


Figure 2.15: Influence on the performance prediction accuracy of the interaction between the various characteristics for a 12-wide 256-entry window processor with a 'large' cache configuration.

2. when annotating statistical branch prediction information and statistical cache miss information to the real trace—in other words, the ILP is taken from the real trace, and the branch and cache behavior are modeled statistically;

3. when annotating the real trace with statistical instruction and data cache miss information—thus assuming real ILP behavior and real branch prediction behavior;

4. when annotating the real trace with statistical branch prediction information—thus assuming real ILP information and real cache behavior.

We can take several interesting conclusions from the results presented in Figure 2.15, namely that:

- statistically modeling ILP introduces a significant error for the SPECint95 traces. This can be concluded from the fact that the average error for the SPECint95 traces is reduced from 12.7% (case 1) to 8.0% (case 2) by using real ILP information instead of statistically generated ILP information. For the IBS traces on the other hand, statistically modeling the ILP does not introduce a significant error since the prediction error is nearly the same in both cases, 14.7% (case 1) versus 13.4% (case 2), respectively.

- statistically modeling branch prediction behavior does not introduce a significant error, except for a few benchmarks such as compress, go and ijpeg. This can be concluded by comparing cases 2 and 3. Not surprisingly, these three benchmarks also show a larger IPC prediction error in Figure 2.8 where we evaluate statistically modeling branch prediction behavior.

- statistically modeling cache miss behavior introduces a significant error for the IBS traces as well as for some SPECint95 benchmarks. Indeed, the prediction error increases for the IBS traces from 1.6% (case 4) to 13.4% (case 2) when modeling the cache miss behavior statistically.

Recall the case of li for which it was hard to find an explanation for the overall prediction error in terms of the various sources. The results in Figure 2.15 confirm this statement and clearly show that the prediction error comes from the interaction between the ILP and the cache behavior, more specifically the D-cache behavior.

2.7.2 Relative accuracy

When you are courting a nice girl an hour seems like a second. When you sit on a red-hot cinder a second seems like an hour. That's relativity.
Albert Einstein

Although absolute accuracy is an important evaluation criterium for performance prediction methods, we believe that relative accuracy is even more important. I.e., when computer architects can make use of accurate estimations of performance trends as a function of microarchitectural parameters, appropriate design decisions can be based upon them. For example, when the performance gain due to increasing a particular hardware resource does not justify the increased hardware cost, designers will decide not to increase that hardware resource. Another interesting application of relative performance prediction is when it is used in combination with early floorplanning. Narayananan et al. [95] present PEPPER, an early floorplanner that is successfully used within IBM for CPU designs to consider timing issues due to floorplanning in the early stages of the design. Consider the case where an early floorplanner determines that in order to obtain a certain clock frequency, additional pipeline stages need to be inserted in the microprocessor pipeline to transport data from one place to another on a chip. These additional stages could be added in the front end or the back end of the pipeline. It is clear that either choice will have a different impact on overall performance. The first option adds to the branch misprediction penalty whereas the second option adds to the L1 D-cache load access time. Obviously, this is an important design decision to be made.

In this section, we will evaluate the relative accuracy of statistical simulation as a function of several microarchitectural parameters: issue width, window size, fetch width, branch misprediction penalty and load execution latency (or L1 D-cache access time). In these experiments, we assume a 'large' cache configuration. The data obtained through real trace simulation are labeled 'real'; the data obtained through statistical simulation are labeled 'estimated'.

Next to displaying raw IPC numbers as a function of an architectural parameter, we also show the *derivative IPC* or the slope of the raw IPC curve. The derivative IPC is particularly interesting because it measures whether the estimated IPC curve 'behaves like' the real IPC curve

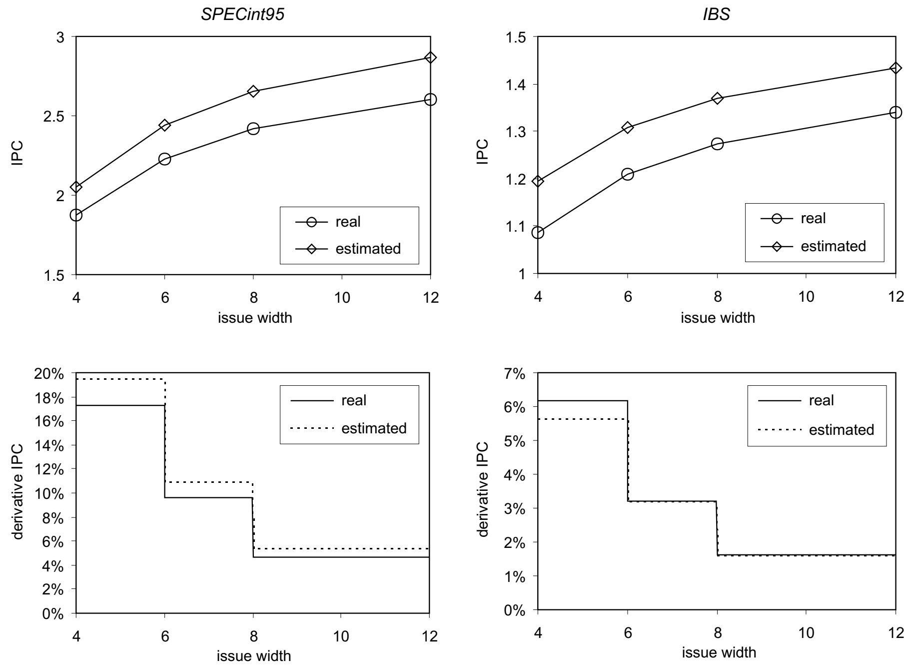


Figure 2.16: IPC and derivative IPC obtained through real trace simulation and statistical simulation as a function of the issue width. These results are average numbers over the benchmark suites.

as a function of an architectural parameter. Indeed, this could be useful while doing design space explorations. If we want to improve the performance of a microarchitectural configuration, this could be done for example by increasing the number of entries in the instruction window, by increasing the issue width, by increasing the fetch width, by decreasing the branch misprediction penalty, etc. If statistical simulation would be able to estimate accurately the slope in each direction in the (multidimensional) design space, correct design decisions are made efficiently.

Issue width. Figure 2.16 shows IPC numbers and derivative IPC numbers as a function of the issue width for real trace simulation and statistical simulation. Note that as we scale the issue width in these experiments, we equally scale the fetch width, the number functional units and the reorder width. From these graphs we can conclude that

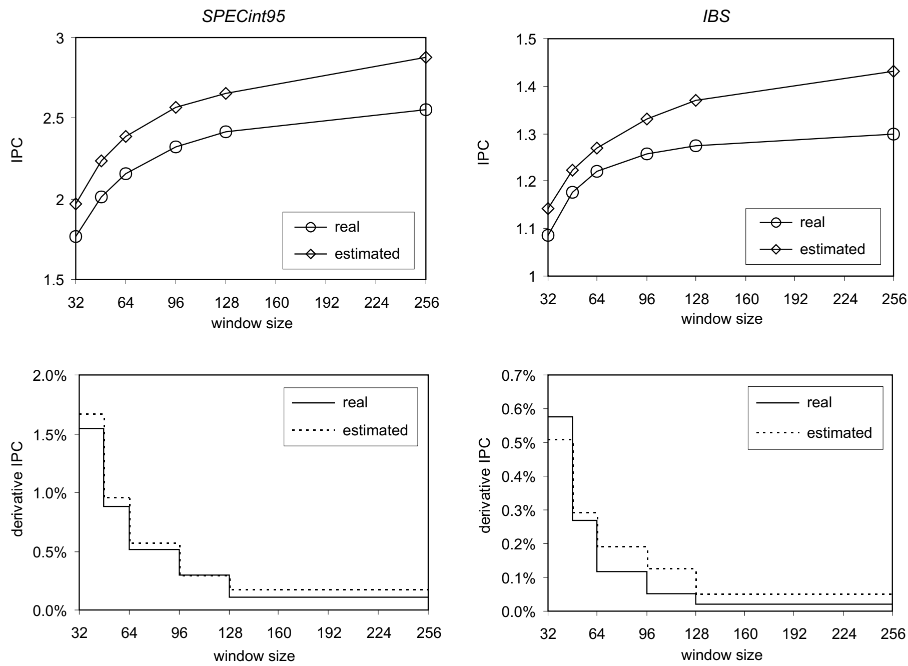


Figure 2.17: IPC and derivative IPC obtained through real trace simulation and statistical simulation as a function of the window size. These results are average numbers over the benchmark suites.

statistical simulation is able to accurately predict the trend as a function of the issue width.

Window size. Figure 2.17 shows IPC numbers and derivative IPC numbers as a function of the window size for real trace simulation and statistical simulation. As can be seen from these graphs, the IPC curve is predicted accurately as a function of window size. Although the slope is severely overestimated by statistical simulation in case of large instruction windows, we do not consider this as a minus of the statistical simulation method because the slope is very small (less than 0.2%).

Fetch width. Figure 2.18 shows IPC numbers and derivative IPC numbers as a function of the fetch width for real trace simulation and statistical simulation. The performance trend is not accurately pre-

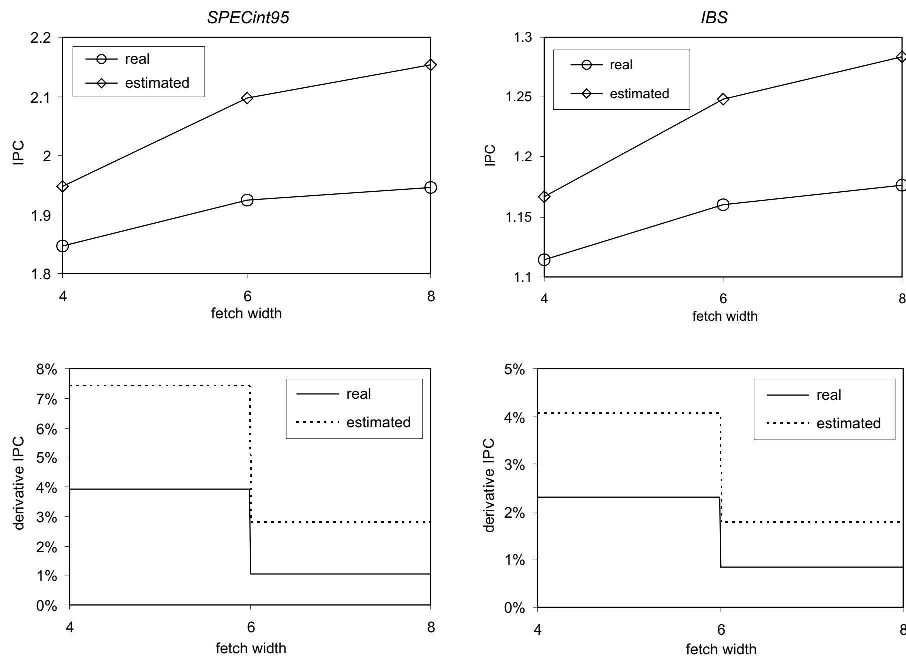


Figure 2.18: IPC and derivative IPC obtained through real trace simulation and statistical simulation as a function of the fetch width. These results are average numbers over the benchmark suites.

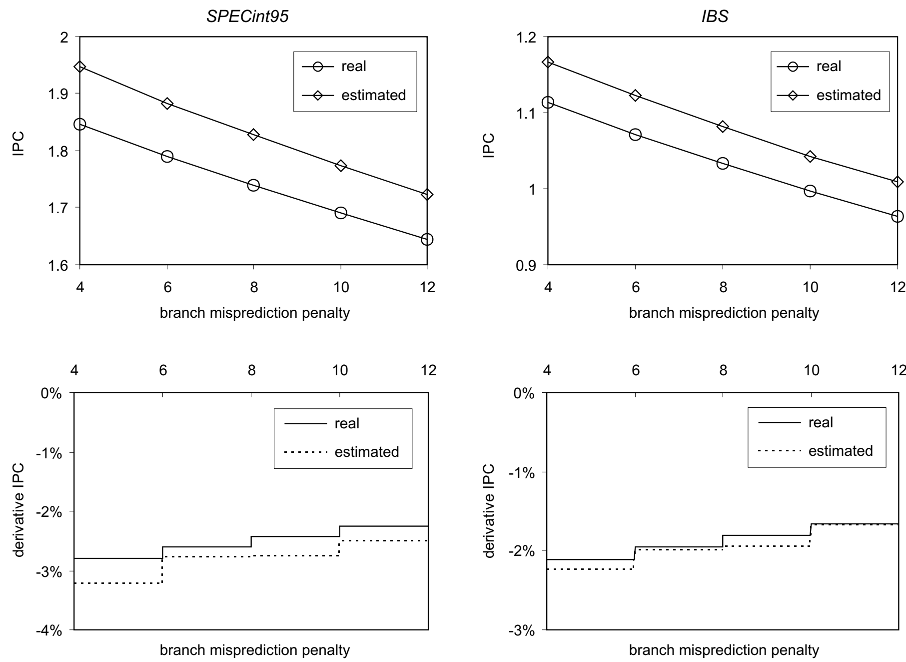


Figure 2.19: IPC and derivative IPC obtained through real trace simulation and statistical simulation as a function of the branch misprediction penalty. These results are average numbers over the benchmark suites.

dicted through statistical simulation. The reason for this phenomenon is the same as why the I-cache statistics presented so far lead to IPC overestimations, as discussed previously. Modeling a more bursty I-cache behavior will result in a better performance trend prediction, as will be shown in the next chapter.

Branch misprediction penalty. Figure 2.19 shows IPC numbers and derivative IPC numbers as a function of the branch misprediction penalty for real trace simulation and statistical simulation by varying the branch misprediction penalty from 4 to 12 cycles. This was done in our simulator by increasing the number of pipeline stages before the dispatch stage. This graph shows that statistical simulation predicts the performance trend very accurately.

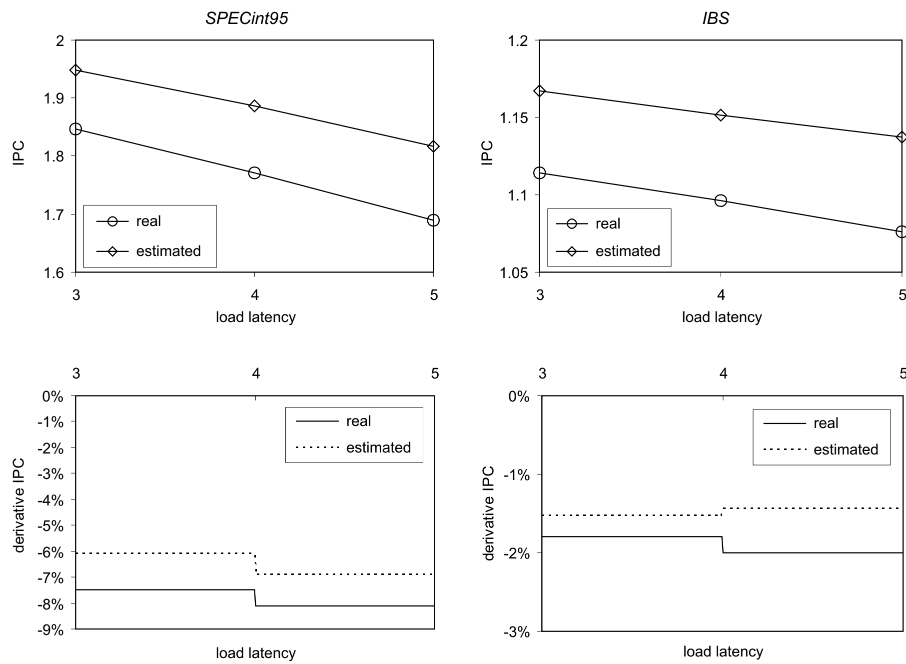


Figure 2.20: IPC and derivative IPC obtained through real trace simulation and statistical simulation as a function of the load latency. These results are average numbers over the benchmark suites.

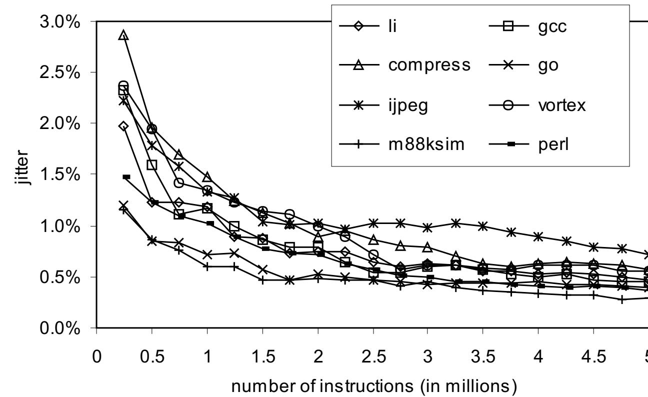


Figure 2.21: Jitter after simulating i synthetically generated instructions for the SPECint95 traces; case of a 128/8 processor with a 'large' cache configuration.

Load latency. Figure 2.20 shows IPC numbers and derivative IPC numbers as a function of the load latency for real trace simulation and statistical simulation. The load latency was set to 3, 4 and 5 cycles; this corresponds to L1 D-cache access latencies of 2, 3 and 4 cycles, respectively. These data confirm that the performance degradation due to an increased load latency is predicted quite accurately.

2.8 Simulation speed

Speed gets you nowhere if you're headed in the wrong direction.
American proverb

In this section, we show that statistical simulation indeed is a fast simulation technique. The performance characteristics quickly converge yielding performance estimates after simulating a few million synthetically generated instructions.

To study this, we have done the following experiment. We have generated 25 different synthetic traces using different random seeds for each original trace. The IPC was measured for all these traces as a function of the number of instructions simulated. For this purpose, we

define the *jitter* as $2 \cdot s_i/\bar{x}_i$, with $\bar{x}_i$ and s_i the average IPC and the standard deviation on the IPC, respectively, after simulating i synthetically generated instructions. The rationale behind this definition is that if we assume that x_i is normally distributed (the Kolomogorov-Smirnov test as well as the Shapiro-Wilks' W test [126] confirm that this is acceptable for nearly all x_i and nearly all benchmarks), the probability that the IPC of a synthetic trace lies in the interval $[\bar{x}_i - 2 \cdot s_i, \bar{x}_i + 2 \cdot s_i]$ is about 95%. In other words, the jitter quantifies the convergence speed of the statistical simulation technique. Figure 2.21 plots the jitter as a function of the number of synthetic instructions simulated for the SPECint95 traces; similar results were obtained for the IBS traces. This graph shows that the jitter is less than 0.75% after simulating 5 million instructions. The results presented in Figure 2.21 are for a 128/8 processor with a 'large' cache configuration. These results justify our statement that statistical simulation is a fast simulation technique since the jitter is quite low after simulating a few million instructions. This corresponds to simulation times of a few minutes instead of several hours or even days in case of a real trace simulation.

2.9 Summary

In this chapter we have presented the general framework of the statistical simulation methodology consisting of statistical profiling, synthetic trace generation and synthetic trace simulation. During statistical profiling a distinction is made between microarchitecture-independent and microarchitecture-dependent program characteristics. The microarchitecture-independent characteristics deal with the instruction mix and the inter-operation dependencies (ILP); the microarchitecture-dependent characteristics deal with branch behavior and cache behavior.

Concerning the synthetic trace generation, we have made an interesting contribution by investigating the implications of guaranteeing syntactical correctness, i.e., store, branch, jump and return operations should not have a destination operand. We found that by guaranteeing syntactical correctness the statistical profile of the synthetic trace does not match the statistical profile of the original trace. We have proposed a mechanism to remedy this: implementing a feedback loop in the synthetic trace generator that monitors the statistical profile of the synthetic trace being generated. At the same time, the synthetic trace

generator adjusts the statistical profile accordingly.

We have also evaluated the accuracy of statistical simulation. We found that the absolute IPC prediction error is generally no larger than 15% to 20%. The major sources of error are the modeling of the ILP and the modeling of cache behavior; statistically modeling branch behavior is found to be have a minor impact on the prediction accuracy. Concerning the relative accuracy we can state that statistical simulation attains good relative accuracy. Finally, we have shown that statistical simulation indeed is a fast simulation technique.

Chapter 3

Increasing the accuracy of statistical simulation

There are many ways of going forward, but only one way of standing still.
Franklin D. Roosevelt

In this chapter, we propose and evaluate two possible improvements to the statistical simulation methodology: (i) using higher-order ILP distributions and (ii) modeling clusters of cache misses. At the end of this chapter, we extensively discuss related work in the area of early design stage performance estimation techniques.

3.1 Higher-order ILP distributions

In the previous chapter, see section 2.4.1, we discussed how we model microarchitecture-independent characteristics, i.e., instruction latencies and inter-operation dependencies. This was done by means of three distributions[1]:

- the instruction mix distribution, or $P\,[\,T_x = t\,]$ with t one of the N_{instr} instruction types;

[1] In this section, we do not consider the distribution concerning memory operations; as such, the age of memory instances distribution is not mentioned in this enumeration.

- the number of operands distribution, or $P\,[\,O_x = o \,|\, T_x = t\,]$ with o the number of register operands; and
- the age of register operands distribution, or $P\,[A_{i,x} = \delta \mid T_x = t, O_x = o]$ with δ the age of the i-th register operand of instruction x.

In this section, we study if using higher-order distributions yields more accurate performance predictions. The approach taken in this section is to measure the above distributions conditionally on the types of the n instructions before instruction x in the trace. Note that $n = 0$ corresponds to the distributions considered so far. The resulting distributions then are, e.g., for $n = 2$:

- $P\,[\,T_x = t \,|\, T_{x-1} = t', T_{x-2} = t''\,]$, the instruction mix distribution;
- $P\,[\,O_x = o \,|\, T_x = t, T_{x-1} = t', T_{x-2} = t''\,]$, the number of operands distribution; and
- $P\,[\,A_{i,x} = \delta \mid T_x = t, O_x = o, T_{x-1} = t', T_{x-2} = t''\,]$, the age of register operands distribution;

with t' and t'' the instruction types of the two instructions before instruction x. As such, we try to model the clustered occurrence of instruction types in instruction traces, e.g., load instructions tend to be clustered at the beginning of a procedure for dealing with procedure arguments saved on the stack.

As discussed in the previous chapter, see section 2.4, the amount of storage required to store a statistical profile should be limited. The total number of probabilities to be stored here is $\mathcal{O}(N_{max} \times N_{instr}^{n+1})$, with in our case $N_{max} = 512$ and $N_{instr} = 13$, which is exponential in n. However, for small values of n—in our case, n ranges from 0 to 3—this is quite acceptable since the number of instruction classes $N_{instr} = 13$ is small as well.

In Figure 3.1, the IPC prediction error is shown when using these higher-order distributions to model the amount of ILP for different values of n, with n ranging from 0 to 3. This is done for three processor configurations: 32/4, 64/8 and 128/16; further, perfect caches and perfect branch prediction are assumed in these measurements. For the two smallest processor configurations, namely 32/4 and 64/8, the average IPC prediction error decreases with increasing values of n. For example for the 32/4 configuration, the average errors are 6.1% ($n = 0$), 5.3%

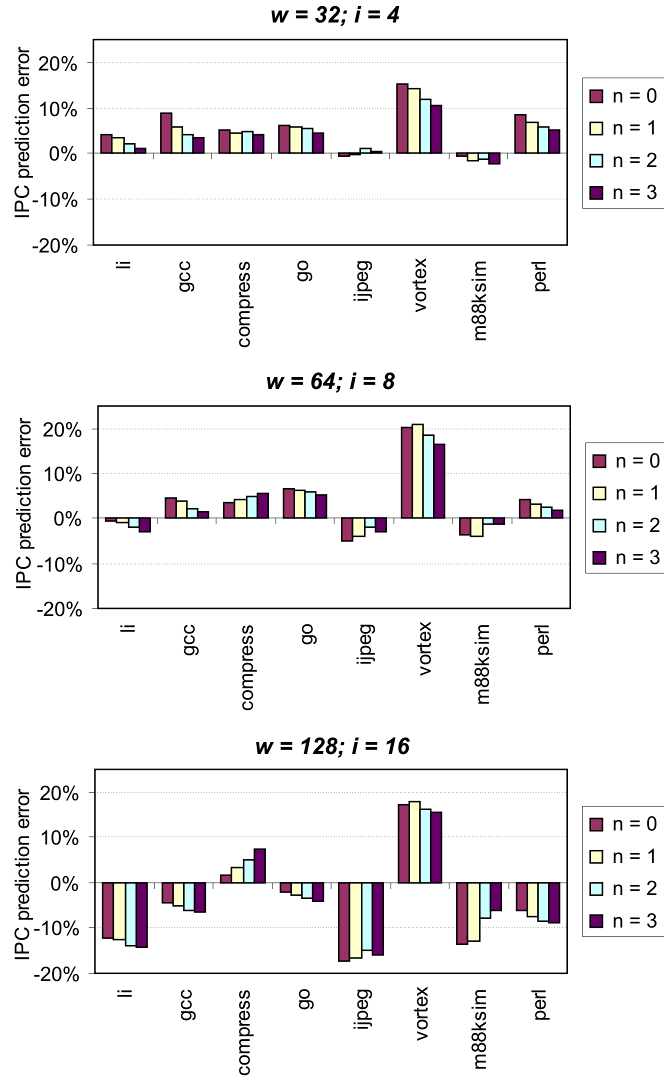


Figure 3.1: IPC prediction errors when using higher-order ILP distributions for three processor configurations: 32/4, 64/8 and 128/16; perfect caches and perfect branch prediction are assumed.

($n = 1$), 4.6% ($n = 2$) and 4.0% ($n = 3$). For the widest processor configuration on the other hand, the average prediction error does not decrease with increasing values of n: 9.3% ($n = 0$), 9.9% ($n = 1$), 9.5% ($n = 2$) and 9.8% ($n = 3$). As such, we can conclude that higher-order ILP distributions do not always increase the prediction accuracy. This result might seem unexpected. However, Nussbaum and Smith [102] also report similar observations in which an improved statistical profile does not always lead to improved performance prediction accuracies. This suggests that the information that is added to the statistical profile is not sufficient to improve the prediction accuracy or is irrelevant for accurate performance prediction. By consequence, such 'improvements' to a statistical profile might lead to more accurate predictions for some benchmarks and to less accurate predictions for others.

3.2 Clustered cache misses

Fractal geometry will make you see everything differently.
There is a danger in reading further.
Michael F. Barnsley

In the previous chapter, we presented simple cache statistics—L1 and L2 I-cache miss rates and L1 and L2 D-cache miss rates. These simple statistics resulted in significant prediction errors, up to 19.3% for the I-cache and up to 25.2% for the D-cache, see section 2.7.1. One of the reasons is that these simple statistics result in a cache miss behavior that is uniformly distributed over the entire execution of the synthetic trace. This is shown in the middle of Figure 3.2. In this figure, the average I-cache miss rate is shown for vortex as a function of time (measured in the number of instructions executed) over various time scales. This was done by averaging the cache miss rate over m consecutive instructions. Although the cache miss behavior shows some burstiness for small values of m, this behavior is averaged out for larger values of m. In addition, the burstiness is not that pronounced for small values of m as it is the case for the cache miss behavior of real programs (compare the left column to the middle column in Figure 3.2). This bursty or *fractal* cache behavior of real programs was reported in previous work by Voldman, Mandelbrot *et al.* [134] and Thiébaut [129].

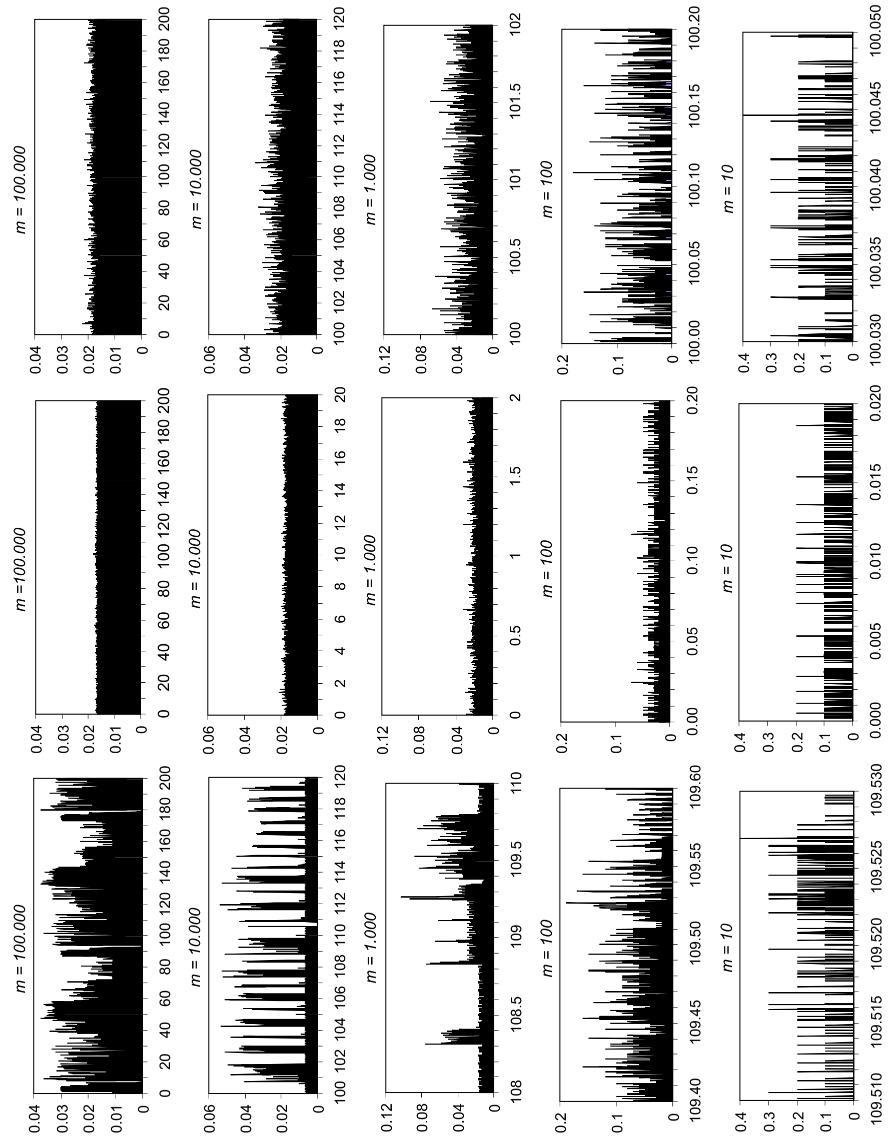


Figure 3.2: On the left: I-cache behavior of a real program, in this casus vortex; in the middle: statistically modeling I-cache behavior using a uniform distribution; on the right: statistically modeling I-cache behavior by modeling intermiss gaps.

3.2.1 Clustered I-cache misses

Not modeling this bursty behavior certainly has its implications on the performance prediction accuracy as we discussed in the previous chapter, see section 2.7.1. To overcome these performance prediction inaccuracies, we propose to model *intermiss gaps*. The statistics introduced to accomplish this, will be called *enhanced* cache statistics in this chapter in contrast to the *simple* cache statistics discussed so far. We have measured the following distributions: $P\ [\ U = u, M_n = L \mid M_p = L\ ]$ with U the intermiss gap, M_n the cache level at which the next cache miss will occur, M_p the cache level at which the previous cache miss has occurred and $L \in \{L1, L2\}$ the level in the cache hierarchy of the cache miss. This distribution quantifies the probability that the next I-cache miss will occur at the M_n-th cache level and that this I-cache miss will be initiated by an instruction that comes U instructions ahead in the trace, given the fact that the previous I-cache miss was a M_p-th cache level miss. Using these enhanced cache statistics to generate synthetic traces yields a cache miss behavior that is significantly different from the behavior using the simple cache statistics. This is shown in the right column of Figure 3.2. For values of $m \leq$ 1,000 (recall that in Figure 3.2 cache miss rates are averaged over m consecutive instructions), the clustered cache miss behavior resembles the cache miss behavior of the real trace. For larger values of m however, the cache miss behavior is averaged over the entire execution of the synthetic trace. This is due to the fact that the autocorrelation of the intermiss gaps is not modeled, i.e., consecutive intermiss gaps are assumed to be statistically independent. This does not harm the performance prediction accuracy as long as processor architectures with instruction windows smaller than 1,000 entries are being modeled, which includes all contemporary and near future designs.

These enhanced I-cache statistics yield more accurate performance predictions, as is shown in Figures 3.3 and 3.4 for the 'small' and the 'large' cache configuration, respectively. In these figures, the IPC prediction error is shown between the IPC obtained from real trace simulation when considering real I-cache behavior (i.e., non-perfect I-cache) and the IPC obtained for the real trace with synthetically generated I-cache miss behavior for the simple and the enhanced characteristics; a perfect D-cache and perfect branch prediction were assumed in all these measurements. Note that the data for the simple cache characteristics are identical to the data presented in Figures 2.9 and 2.10 for

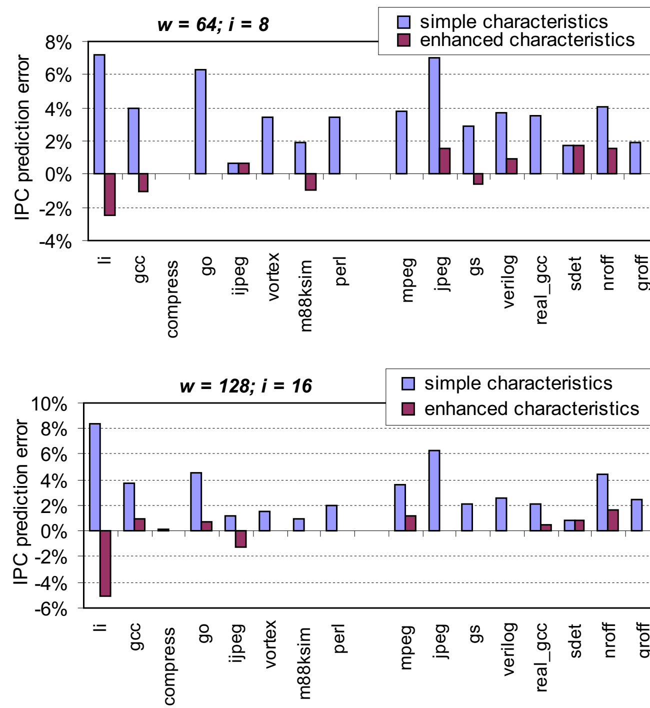


Figure 3.3: Enhanced modeling of I-cache behavior: case of 'small' cache configuration.

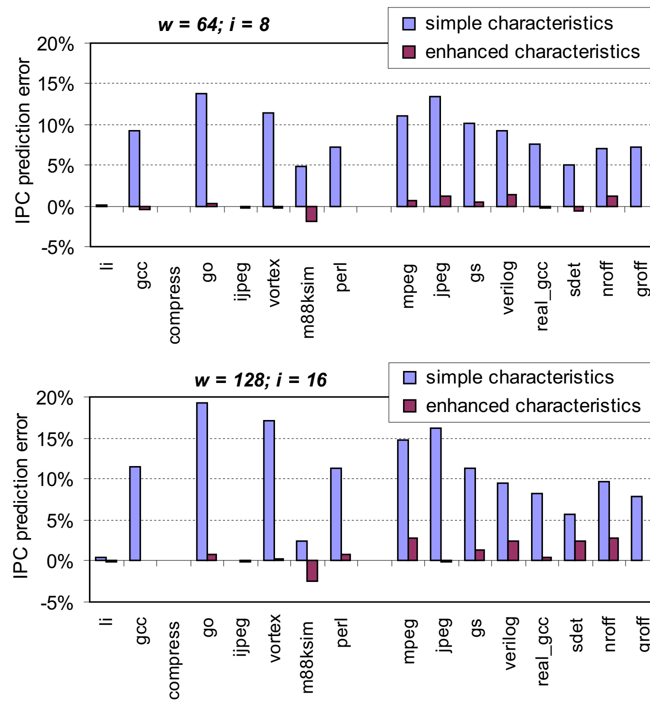


Figure 3.4: Enhanced modeling of I-cache behavior: case of 'large' cache configuration.

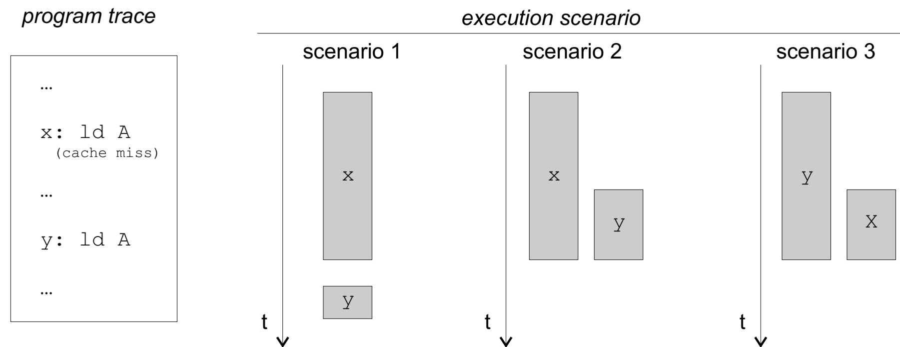


Figure 3.5: The impact on performance of delayed hits.

the 'small' and the 'large' cache configuration, respectively. From these results, we can conclude that modeling clusters of I-cache misses indeed implies better performance prediction accuracies; the maximum prediction error is drastically reduced from 19.3% to 5.0%.

3.2.2 Clustered D-cache misses

We have measured similar distributions (for loads as well as for stores) to model the bursty D-cache miss behavior. To be able to model the impact on performance of accesses to data words to the same (32-byte) cache block that causes a cache miss, so called *delayed hits* or *secondary misses* [52, 81], we have measured additional distributions, namely the probability that a memory operation that comes δ instructions ahead in the instruction trace accesses the same cache block as the memory operation that initiated a cache miss. Consider the case where a load needs to be executed in the synthetic trace simulator that accesses the same cache block as a memory operation ahead in the synthetic trace that was annotated to cause a cache mis. Figure 3.5 illustrates what the impact of such a case will be on performance. In Figure 3.5, the cache miss labeled memory operation will be referred to as load `x`; the second load that accesses the same memory address will be referred to as load `y`. This will have the following implications on the execution of these memory instructions depending on the execution scenario:

- scenario 1: load `x` is executed when load `y` is issued. As such, load `y` gets the L1 D-cache access time assigned;
- scenario 2: load `x` is still executing when load `y` is issued. In this

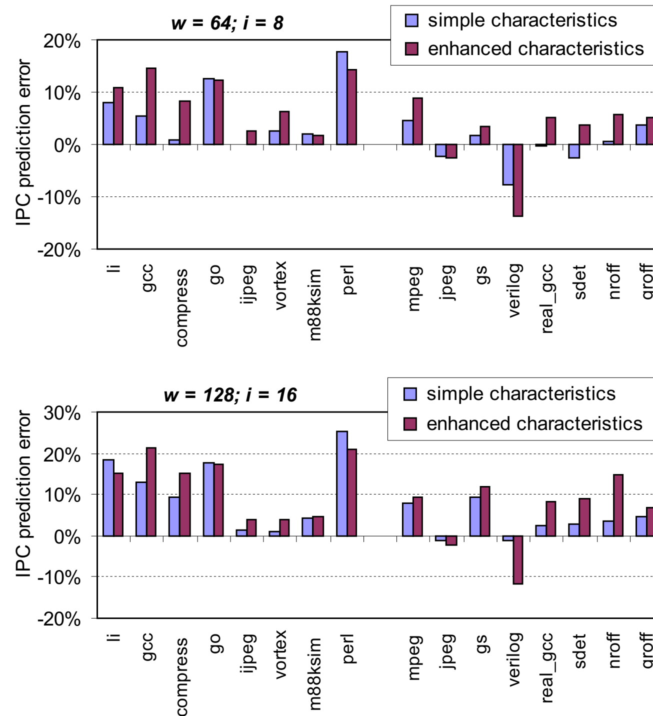


Figure 3.6: Enhanced modeling of D-cache behavior: case of 'small' cache configuration.

case, when load y is issued it will get the remaining execution latency assigned of load x.

- scenario 3: load x is not yet executing when load y is issued. Load y gets the execution latency of a load accessing the L2 D-cache or main memory assigned if load x misses at the L1 or L2 level, respectively. Load x that will be executed later on, will then get the remaining execution latency of the resolving load y.

To evaluate these enhancements, we have set up several experiments of which the results are shown in Figures 3.6 and 3.7 for the 'small' and the 'large' cache configuration, respectively. The IPC prediction error is shown between a real trace simulation with non-perfect D-caches versus a simulation of a real trace annotated with synthetically generated D-cache miss behavior; perfect branch prediction and a perfect I-cache were assumed. The data for the simple characteris-

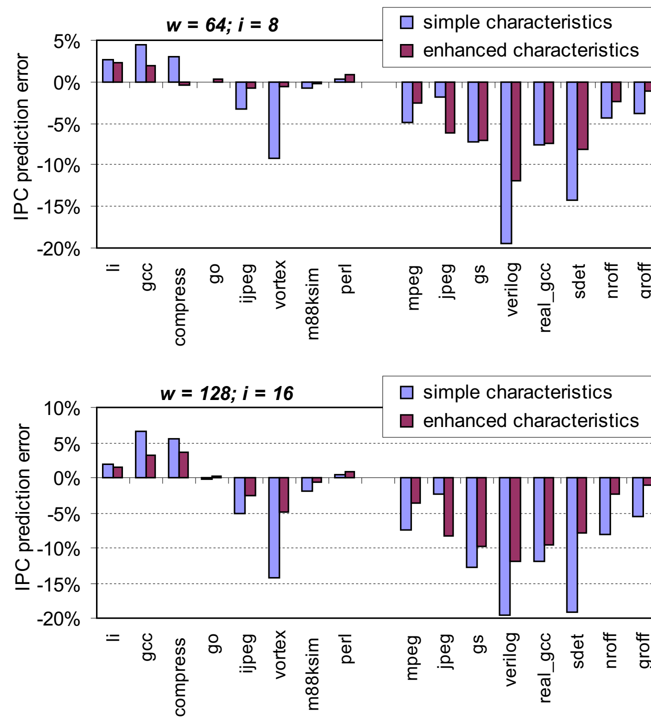


Figure 3.7: Enhanced modeling of D-cache behavior: case of 'large' cache configuration.

tics are the same as from Figures 2.11 and 2.12. The maximum prediction error is reduced from 25.2% to 21.0% for the 'small' cache configuration, and from -19.5% to -12.0% for the 'large' cache configuration. However, these results indicate that the performance prediction accuracy is not always increased by modeling intermiss gaps and delayed hits. Indeed, for the SPECint95 traces with a 64/8 processor and a 'small' cache configuration, the average prediction error even increases from 6.2% to 8.9%. This is due to the fact that some loads are latency tolerant—some loads can tolerate long latencies without degrading performance—while others are not [124]. The fact that a load is latency tolerant is dependent on its dependency graph, i.e., on the number of instructions and their types that are (directly or indirectly) dependent on that load. This could be modeled by taking the dependency graph of loads into account when assigning D-cache misses.

3.2.3 Overall prediction accuracy

Figures 3.8 and 3.9 evaluate the overall prediction accuracy with the enhanced I-cache and D-cache characteristics for the 'small' and the 'large' cache configuration, respectively. These results compare the IPC of a real trace to the IPC of a synthetic trace under the following circumstances:

(i) using simple I-cache and D-cache characteristics, resulting in the same data as presented in Figures 2.13 and 2.14;

(ii) using enhanced I-cache characteristics and simple D-cache characteristics; and

(iii) using enhanced I-cache and D-cache characteristics.

The average prediction errors are tabulated in Table 3.1. The enhanced I-cache characteristics lead to smaller IPC prediction errors in most cases. For example for the IBS traces and a 128/16 processor with a 'small' cache configuration, introducing enhanced I-cache statistics decreases the average error from 13.3% to 6.5%. Adding enhanced D-cache statistics next to enhanced I-cache statistics, further decreases the prediction error to 5.3%. In general, enhanced I-cache statistics lead to a significant reduction in prediction error. However in some cases, adding enhanced D-cache statistics in addition, does not always lead to an increased accuracy, compare cases (ii) and (iii) in Table 3.1.

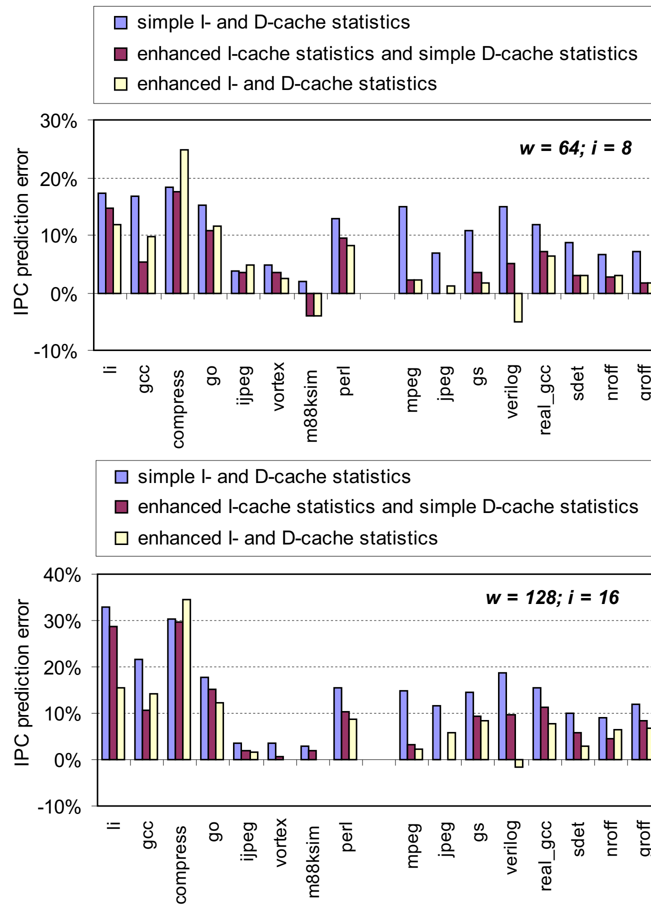


Figure 3.8: Overall prediction accuracy with enhanced modeling of cache behavior: case of 'small' cache configuration; (i) simple I- and D-cache statistics, (ii) enhanced I-cache statistics and simple D-cache statistics and (iii) enhanced I- and D-cache statistics.

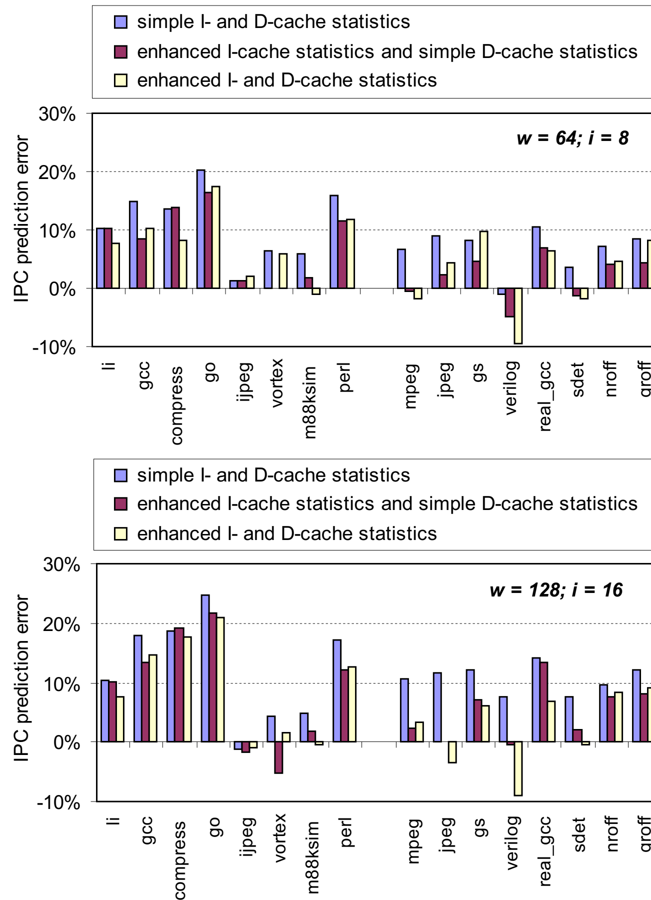


Figure 3.9: Overall prediction accuracy with enhanced modeling of cache behavior: case of 'large' cache configuration; (i) simple I- and D-cache statistics, (ii) enhanced I-cache statistics and simple D-cache statistics and (iii) enhanced I- and D-cache statistics.

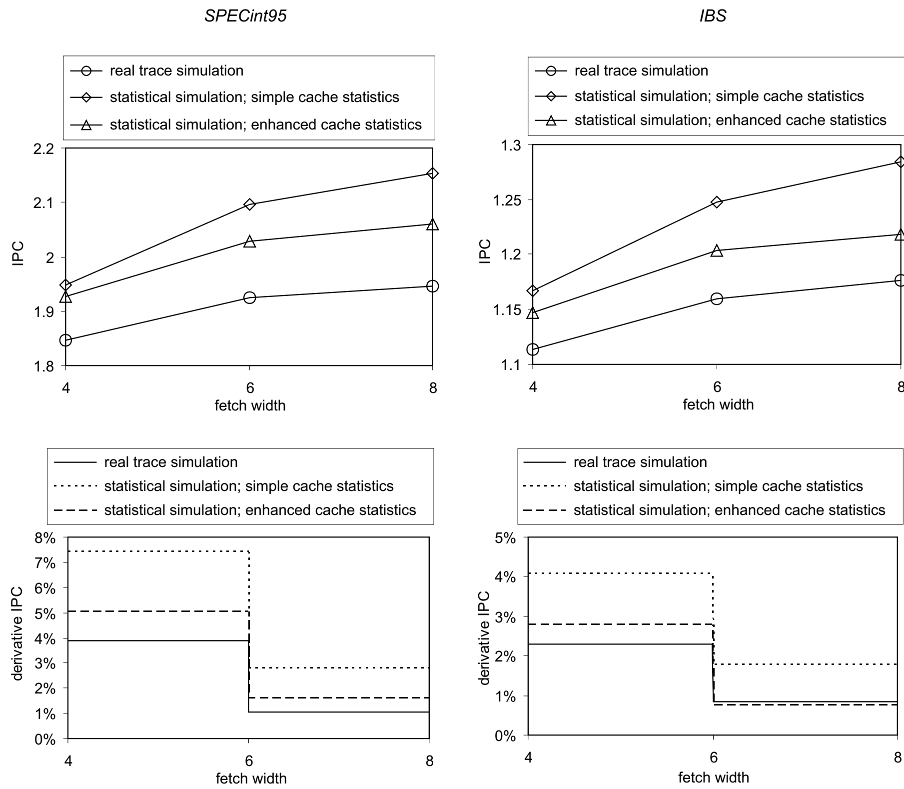


Figure 3.10: IPC and derivative IPC obtained through real trace simulation and statistical simulation when using simple and enhanced cache statistics as a function of the fetch width. These results are average numbers over the benchmark suites.

	'small' cache configuration					
	64/8			128/16		
	(i)	(ii)	(iii)	(i)	(ii)	(iii)
SPECint95	11.4%	8.6%	9.7%	16.0%	12.4%	10.9%
IBS	10.2%	3.2%	3.1%	13.3%	6.5%	5.3%

	'large' cache configuration					
	64/8			128/16		
	(i)	(ii)	(iii)	(i)	(ii)	(iii)
SPECint95	11.0%	8.0%	8.0%	12.4%	10.7%	9.6%
IBS	6.8%	3.6%	5.8%	10.7%	5.1%	5.9%

Table 3.1: Average IPC prediction error with enhanced modeling of cache behavior: (i) simple I- and D-cache statistics, (ii) enhanced I-cache statistics and simple D-cache statistics and (iii) enhanced I- and D-cache statistics.

Relative accuracy. In the previous chapter, we discussed that predicting performance as a function of the fetch width is highly inaccurate when using simple cache statistics, see Figure 2.18 in section 2.7.2. Figure 3.10 presents the performance trend predictions in the following three cases: (i) real trace simulation, (ii) statistical simulation using simple cache statistics—the data in (i) and (ii) are the same as in Figure 2.18—and (iii) statistical simulation using enhanced cache statistics. From these graphs we can conclude that using enhanced cache statistics leads to a more accurate performance trend prediction.

Simulation speed. Although using enhanced cache statistics leads to higher performance prediction accuracies in general, there is also a drawback concerning enhanced cache statistics. The maximum jitter, as defined in section 2.8, increases dramatically from 0.75% when using simple cache statistics to 5.5% after 5 million instructions when using enhanced statistics. The reason for this is that enhanced cache statistics introduce large intermiss gaps (sometimes more than 100,000 instructions) that do not occur in case of simple cache statistics. This sometimes results in large variations in performance characteristics when simulating synthetic traces, which results in such a large jitter. In practice, this means that it could happen that the IPC estimate for configuration B is smaller than the IPC estimate for configuration A, although the hardware resources are larger for B than for A. This situation is imaginable if the real performance gain from A to B is quite small (a few percent). There are two possible ways for addressing this problem.

First, by generating and simulating longer synthetic traces, for example 15 million instructions (jitter = 3%) or 10 million instructions (jitter = 3.6%) instead of 5 million instructions. Second, by averaging the IPC estimates of several shorter synthetic traces, for example by averaging over 5 synthetic traces each containing 2 million instructions. In conclusion, a practical consequence of these enhanced cache statistics is a longer simulation time. Note that the performance results presented in Figures 3.8 and 3.9 were obtained by averaging the performance estimates of several synthetic traces using different random seeds.

3.3 Related work

In a major matter, no details are small.
French proverb

In this section, we discuss related work in the area of reducing the total simulation time. Several approaches will be considered: statistical simulation, analytical performance modeling, trace sampling and a collection of other approaches. We also detail on the relation of the work presented in this dissertation so far to the research efforts published in previous work.

3.3.1 Statistical simulation

Noonburg and Shen [101] presented a framework that models the execution of a program on a particular architecture as a Markov chain, in which the state space is determined by the microarchitecture and in which the transition probabilities between the various states are determined by the program execution. This approach has two major disadvantages. First, when modeling wide-issue superscalar processors, the Markov chain becomes too complex [118]. Second, the statistical profile presented in [101] is too complex to model wide-issue processors. The authors used the following distribution to determine the type T_x and the *dependent instruction distance* D_x of an instruction x: $Prob\,[\,T_x = t\,,\,D_x = \delta \mid T_{x-1} = t'\,,\,D_{x-1} = \delta'\,,\,T_{x-2} = t''\,,\,D_{x-2} = \delta''\,]$. These distributions require $N_{max}^3 \times N_{instr}^3$ probabilities to be stored in a statistical profile. This is feasible for performance modeling of architec-

tures with small instruction windows where N_{max} can be restricted, e.g. $N_{max} = 5$ in [101]. But in our case this would require $512^3 \times 13^3 \approx 295\text{G}$ probabilities to be stored, which is impossible. These disadvantages are overcome in our performance modeling environment. The statistical profile used here is less complex than the one presented in [101] while maintaining high levels of accuracy, and the architecture is modeled as a trace-driven simulator which is much less complex than a corresponding Markov chain. Moreover, Noonburg and Shen [101] did not model memory dependencies. Another important issue is the way inter-operation dependencies are considered: only the youngest age measured in program order of both register operands is used to determine if an instruction is issuable. This is correct for an in-order processor, but is definitively not for an out-of-order architecture. Therefore, both register operands are considered here. Notice that this is also required to guarantee syntactical correctness of the synthetic traces. In their evaluation (while considering perfect caches), Noonburg and Shen [101] report a maximum IPC prediction error of 1% for a single-pipeline in-order processor and a maximum IPC prediction error of 10% for a three-pipe in-order processor.

Hsieh and Pedram [67] present a technique to estimate performance and power dissipation of a microprocessor by first measuring a characteristic profile of a program execution, and by subsequently synthesizing a new, fully functional program that matches the extracted characteristic profile. The characteristic profile includes the instruction mix, branch prediction accuracy, cache miss rate, pipeline stall rate and IPC. The program that is synthesized using this characteristic profile, is then executed on an execution-driven simulator to estimate performance and power consumption. Since the dynamic instruction count of the synthesized program is smaller than the dynamic instruction count of the original program, the simulation time is significantly reduced. The prediction errors for both power dissipation and IPC are less than 5%. The major drawback of this approach is that no distinction is made between microarchitecture-independent and microarchitecture-dependent characteristics; all characteristics are microarchitecture-dependent. Consequently, this approach cannot be used for architectural design space explorations.

The statistical simulation methodology as it is studied in this dissertation, was initially presented by Carl and Smith [20]. They proposed an approach in which a synthetic instruction trace is generated based on execution statistics and is subsequently fed into a trace-driven sim-

ulator. They evaluated the error introduced by the various components in the model, similar to section 2.7.1, and they took the same conclusions as we do, namely that the major sources of error are statistically modeling ILP and cache behavior. Statistically modeling branch behavior on the other hand is inaccurate for only a few benchmarks. The errors on overall performance predictions reported vary between 10% and 80%.

Nussbaum and Smith [102] continued this work and presented a different method for generating inter-operation dependencies. In this dissertation, we generate what they call *upstream dependencies*, i.e., an instruction is made dependent on a preceding instruction. As discussed in section 2.5, this can lead to syntactically incorrect synthetic traces in which stores, branches, jumps or returns have a destination operand. This is remedied here by implementing a feedback loop in the synthetic trace generator which guarantees syntactical correctness as well as an exact implementation of the desired distribution. Nussbaum and Smith [102] on the other hand, use so called *downstream dependencies*, which means that a future instruction is made dependent on the current instruction. As such, the problem with branches and stores having a destination operand is fixed. However, this introduces a new problem, namely that some instructions will have more (or less) source operands than is allowed syntactically. For example, an addition can have five source operands or a store can have zero source operands. In [102], no fix-up is performed in those cases. Nussbaum and Smith also present an evaluation of using various higher-order distributions in which the instruction mix, the inter-operation dependencies, the cache miss rates and the branch misprediction rates are correlated to the basic block size. The authors conclude that these higher-order distributions indeed can lead to higher performance prediction accuracies, e.g., the average IPC prediction error can be reduced from 15% to 9% for a wide-resource 128/8 microprocessor configuration. However, they also report experimental results suggesting that simple statistical models are accurate enough for doing design space explorations.

Nussbaum and Smith continued their work by evaluating symmetric multiprocessor system (SMP) performance through statistical simulation [103]. They evaluated multiprogrammed workloads as well as parallel scientific workloads. For this purpose the statistical simulation methodology is extended to include statistics on cache coherence events, sequential consistency, critical sections, lock accesses and barrier distributions. They conclude that statistical simulation is suffi-

ciently accurate to predict SMP performance trends.

Oskin, Chong and Farrens [106] present the HLS simulation environment which is basically the same as the statistical simulation methodology presented by Carl and Smith [20] and the simple model presented in this dissertation. The work done by Oskin, Chong and Farrens [106] has two major contributions. First, they validate the statistical simulation methodology against real hardware, namely a MIPS R10000 processor, and they conclude that statistical simulation indeed achieves a high performance prediction accuracy (a maximum error of 7.8% is reported). Second, they evaluate how well statistical simulation predicts performance under varying branch prediction accuracies, L1 I-cache miss rates, L1 D-cache miss rates and compiler optimization levels. These experiments are so called single-value correlation studies, i.e., by varying only one parameter in each experiment. They also performed multi-value correlation studies by varying several parameters simultaneously. This kind of experiments is extremely useful for identifying in which area of the design space statistical simulation can be used with confidence.

The work presented in this dissertation so far makes the following contributions to the work done in the area of statistical simulation: (i) proposing a feedback loop in the synthetic trace generator for guaranteeing syntactical correctness of synthetic traces while preserving the representativeness of the synthetic trace (exact implementation of the desired distributions), (ii) emphasizing the importance of the relative accuracy of an early design stage methodology and the evaluation of the relative accuracy of statistical simulation, (iii) evaluating higher-order ILP distributions, and (iv) modeling clustered cache misses which significantly reduces the average IPC prediction error: 7% to 16% when modeling simple cache statistics versus 3% to 11% when modeling enhanced cache statistics.

3.3.2 Analytical modeling

Jouppi [74] show that instruction-level parallelism (ILP) is not uniformly distributed over an entire program execution due to (i) variations in instruction latencies, (ii) dependencies between instructions, and (iii) variations in parallelism by instruction class. This knowledge of non-uniformity is used in this dissertation in our search for viable statistical profiles: inter-operation dependencies are considered for

various instruction classes.

Noonburg and Shen [100] present an analytical model that uses parallelism distributions measured from real program traces, without actually modeling inter-operation dependencies. Moreover, single-cycle instruction latencies as well as no memory dependencies were assumed.

Dubey, Adams and Flynn [29] propose an analytical performance model on the basis of two parameters that are extracted from a program trace. The first parameter, the *conditional independence probability* p_δ, is defined as the probability that an instruction x is independent of instruction $x - \delta$ given that x is independent of all instructions in the trace between x and $x - \delta$. The second parameter p_ω is defined as the probability that an instruction is scheduled with an instruction from ω basic blocks earlier in the program trace. The main disadvantages of this analytical model is that only one dependency is considered per instruction and that no differentation is made between various instruction classes for p_δ, which will lead to inaccurate performance estimates.

Squillante, Kaeli and Sinha [123] propose analytical models to capture the workload behavior and to estimate pipeline performance. Their technique was evaluated for a single-issue pipelined processor.

Sorin *et al.* [122] present a performance analysis methodology for shared-memory systems that combines analytical techniques with traditional simulations to speed up the design process.

Simple analytical models are often used to get insight in the impact of microarchitectural parameters on performance [50, 54, 61, 92, 98].

3.3.3 Trace sampling

Another widely used method for speeding up the simulation process is trace sampling. Trace sampling was first proposed in the context of cache simulation [83, 127], but was later extended for full processor simulation [23]. A *sampled trace* is obtained from an original program trace by gathering samples from the original trace. There are two possible ways of selecting samples: one can take one single large sample, e.g., one sample of 50 million references [116, 117], or one can take multiple (small) samples, e.g., 50 evenly spaced samples each containing 1 million references [30, 71, 72, 78, 82, 84, 88, 108]. The *sample rate* is defined as the ratio of the number of references in the sampled trace divided by the number of references in the original trace. An important problem with trace sampling is the hardware state (caches, branch

predictor, processor core, etc.) at the beginning of each sample. This problem is well known as the *cold-start problem* [24, 25, 62, 76, 99, 137]. One possible way of dealing with the cold-start problem is to simulate (without computing performance characteristics) additional references before each sample to warm-up hardware structures, such as branch predictors and caches. These additional references are part of the *warm-up*. The *simulation speedup* as a result of trace sampling is the ratio of the number of references in the original trace divided by the number of references in the sampled trace plus the number of references in the warm-up. Generally, a higher sample rate leads to a higher accuracy at the cost of a lower simulation speedup.

Although we did not perform a comparison between statistical simulation and trace sampling we can make the following considerations. It is our intuition that trace sampling can achieve a higher accuracy, however, at the cost of a longer simulation time. On the other hand, for the same simulation speedup, we can expect statistical simulation to be at least as accurate as trace sampling. An important benefit of trace sampling compared to the current state of statistical simulation is that trace sampling can take into account the temporal behavior of the program execution [116] by taking samples from various program phases[2]. On the other hand, statistical simulation has the important advantage over trace sampling that a profile of the *complete* program execution can be characterized in a cost-effective way, whereas trace sampling requires additional samples which will ultimately reduce the simulation speedup. In conclusion, a fair comparison between trace sampling and statistical simulation is an interesting subject for future research. However, we believe that statistical simulation is more appropriate in the earliest stages of the design for doing efficient design explorations, i.e., for identifying an interesting area in the design space. Trace sampling on the other hand, can then be used in a subsequent phase of the design in which more detailed and thus slower simulation runs are justified by the higher accuracy of the technique and the limited design space.

3.3.4 Other approaches

Next to statistical simulation, analytical modeling and trace sampling, there are a number of other interesting approaches published in the

[2]A similar approach could be taken for statistical simulation by taking different statistical profiles for the various program phases.

literature to speedup the simulation process. These other approaches are further subdivided in two categories: model-based approaches and engineering-based approaches.

Model-based approaches. Ofelt and Hennessy [105] present a profile-based performance prediction technique that is an extension of a well known approach consisting of two phases. The first (instrumentation) phase counts the number of times a basic block is executed. The second phase then simulates each basic block while measuring the amount of IPC. This estimated IPC number is multiplied by the number of times the basic block is executed during a real program execution. The sum over all basic blocks then gives an estimate of the IPC of program. The approach presented by Ofelt and Hennessy [105] extends this simple method to enable the modeling of out-of-order architectures, e.g., by modeling parallelism between instructions from various basic blocks. This approach achieves a high accuracy (errors of only a few percent are reported) while assuming perfect branch prediction and perfect caches. However, when a realistic branch predicor and realistic caches are included in the evaluation, the accuracy falls short [104].

Loh [85] presents a time-stamping algorithm that achieves an average accuracy of 7.5% with a 2.42X simulation speedup for wide-issue out-of-order architectures. This approach is built on the idea that it is sufficient to know *when* events—such as the end of an instruction execution or the availability of a resource—occur. By time-stamping the resources associated with these events, the IPC can be computed by dividing the number of instructions simulated by the highest time stamp. The inaccuracy comes from making assumptions in the time-stamping algorithm which make it impossible to accurately model the behavior of a complex out-of-order architecture such as out-of-order cache accesses, wrong path cache accesses, etc.

Brooks, Martonosi and Bose [15] evaluate the popular abstraction via separable components method which considers performance as the summation of a base performance level (idealized base cycles per instruction or CPI while assuming perfect caches, perfect branch prediction, etc.) plus additional stall factors due to conflicts, hazards, cache misses, mispredictions, etc. A simulation speedup is obtained with this technique since the base performance level and the stall factors can be computed using simple simulators instead of fully-detailed and thus slower simulators. They conclude that for modeling out-of-order archi-

tectures, this methodology attains, in spite of its poor absolute accuracy, a reasonable relative accuracy.

Engineering-based approaches. KleinOsowski, Flynn, Meares and Lilja [79, 80] propose to use reduced input sets for the SPEC2000 benchmark suite [65] which results in significantly lower simulation times compared to the reference input sets provided by the Standard Performance Evaluation Corporation (SPEC)[3]. This approach will be further discussed in chapter 6. Haskins, Skadron, KleinOsowski and Lilja compare trace sampling versus reduced input sets in [63].

Bose [10] proposes to *pre-process* a program trace, e.g., by tagging loads and stores with hit/miss information, or by tagging branches with prediction information (wrong or correct prediction). This tagged program trace can then be executed on a simulator that imposes an appropriate penalty in the simulator. A similar approach was taken in section 2.7.1 to evaluate the various components in the statistical simulation method.

Schnarr and Larus [115] show how to speed up an out-of-order processor simulator using memoization. Traditionally, memoization refers to caching function return values in functional programming languages. These cached values can then be returned when available during execution avoiding expensive computations. Schnarr and Larus present a similar technique that caches microarchitecture states and the resulting simulation actions. When a cached state is encountered during simulation, the simulation is then fast-forward by replaying the associated simulation actions at high speed until a previously unseen state is reached. They achieve an 8 to 15 times speedup over SimpleScalar's out-of-order simulator [18] while producing exactly the same result.

Lauterbach [84] and Nguyen *et al.* [99] speed up the simulation process by cutting a program trace in trace chunks, and simulating these chunks in parallel on a cluster of workstations. As with trace sampling, an important problem that needs to be solved is the correct state at the beginning of each trace chunk.

[3] `http://www.spec.org`

3.4 Summary

In this chapter we have evaluated two possible improvements to the statistical simulation methodology. First, we have evaluated whether higher-order ILP distributions that are correlated on the instruction types of the previous n instructions in the trace yield an improved accuracy. We found that this does not always lead to an improved accuracy.

Second, we propose to use distributions of intermiss gaps to model clustered cache behavior which reduces the average prediction error from the range 7%–16% to the range 3%–11%. We found that this significantly improves the modeling of the I-cache miss behavior. For the D-cache behavior on the other hand, modeling intermiss gaps and delayed hits does not always lead to an improved accuracy. A possible explanation is the notion of load latency tolerance that is not modeled in the current status of statistical simulation. The improved accuracy due to better I-cache behavior modeling comes at the cost of a slower simulation speed, i.e., longer convergence time.

Finally, we also extensively discussed related work in the field of early design stage performance estimation techniques, such as statistical simulation, analytical modeling, trace sampling and a number of other approaches.

Chapter 4

Register traffic characteristics

The only source of knowledge is experience.
Albert Einstein

By inspecting the statistical profiles that were obtained in the previous chapters, we observed that the inter-operation dependency distributions exhibit power law properties. This is extensively analyzed in this chapter. An interesting application of this property is that it can be used in a hybrid analytical-statistical model that is nearly as accurate as the statistical simulation method from the previous chapters. In addition, this analytical workload model can be used for doing workload space explorations by varying the various parameters included in the model.

4.1 Register traffic characteristics

In modern out-of-order architectures, the registers are the primary means for inter-operation communication. When an operation writes a value into a register, a new *register instance* is *created*. Operations that read a register, are said to *use* the corresponding register instance. Franklin and Sohi [55] proposed four metrics to characterize the register traffic in modern out-of-order architectures. These metrics are measured on a dynamic instruction stream. The first metric is the *degree of use of register instances* which measures the number of times a

```
...
add r1,r1 -> r2
add r2,r1 -> r3
ld r3,r2 -> r4
add r4,r5 -> r2
...
```

Figure 4.1: This example illustrates the register traffic characteristics. Consider four instructions taken from a dynamic instruction stream. Register `r2` is written by the first operation and read by the second and the third. This results in a *degree of use* of 2 and an *age* of 1 and 2. The same register `r2` is written again by the fourth operation. As such, the *lifetime* of this register instance is 3. And since the last use of this register instance is by the third operation, the *useful lifetime* is 2.

register instance is used by other operations. The three remaining metrics quantify the *temporal locality of creation and use of register instances*: (i) the *age of register instances*, i.e., the number of (dynamic) operations between the use and the creation of a register instance; (ii) the *useful lifetime of register instances*, i.e., the number of operations between the creation and the last use of a register instance; and (iii) the *lifetime of register instances*, i.e., the number of operations between two consecutive writes to the same register. An example is shown in Figure 4.1 to illustrate the definition of the register traffic characteristics.

We have measured these four metrics for the SPECint95 benchmarks and the IBS traces mentioned in the previous chapter. The probability distribution functions (PDFs) shown in Figures 4.2 and 4.3 are averaged over the SPECint95 benchmarks and the IBS traces, respectively. We excluded the zero register from all these measurements[1]. These PDFs clearly follow a straight line when displayed in a log-log diagram. A PDF that follows a straight line in a log-log diagram is called a *power law* distribution or a *Pareto* distribution with probability distribution function $P\left[X = x\right] = \alpha x^{-\beta}$ for which $1 > \alpha > 0$ is the intersect of the PDF with the Y axis and $\beta > 0$ is the slope of the straight line of the PDF in a log-log diagram.

It is important to emphasize that we observe this power law behavior for two sets of instruction traces coming from two different workloads, namely SPECint95 and IBS, with different compilers and for dif-

[1] r31 for the SPECint95 benchmarks on Alpha and r0 for the IBS traces on MIPS.

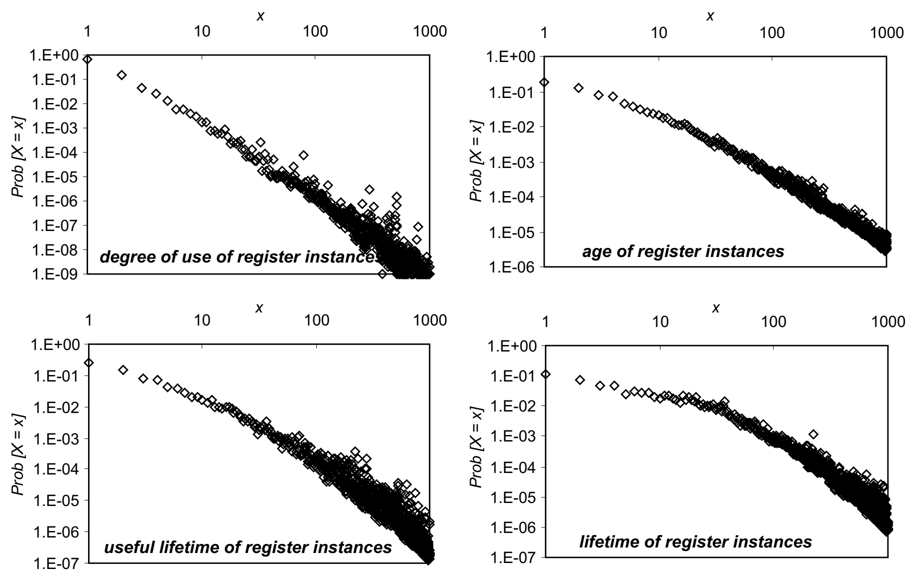


Figure 4.2: Register traffic metrics averaged over the SPECint95 benchmarks: (i) degree of use, (ii) age, (iii) useful lifetime and (iv) lifetime of register instances.

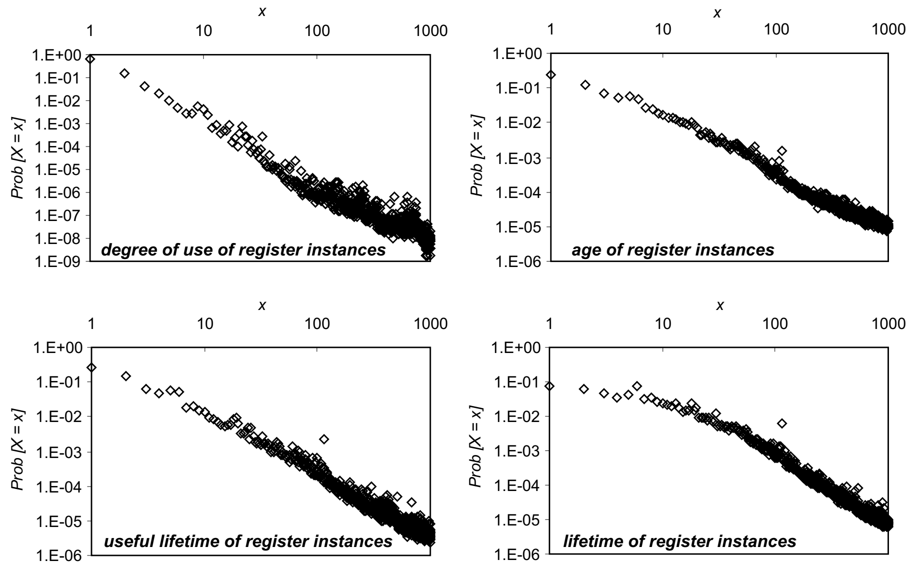


Figure 4.3: Register traffic metrics measured over the IBS traces: (i) degree of use, (ii) age, (iii) useful lifetime and (iv) lifetime of register instances.

ferent instruction set architectures (ISAs), namely Alpha and MIPS, respectively. This strengths us in our statement that this power law behavior will be observed over a wide range of programs with different compiler and ISA settings.

Where does this behavior come from? To answer this question we have done a measurement on the SPECint95 traces (for the Alpha ISA) which reveals that the longest ages of register instances can be caused by the following mechanisms. E.g.,

- for li, 49% of the ages of register instances higher than 100 are caused by the global data section pointer or register r29. The global data section pointer is used to point to the statically allocated data in a computer program. This register is typically written at the beginning of a function or a program but remains unchanged throughout the function execution or program execution, respectively, resulting in long ages.

- for go, 56% of the ages higher than 100 are caused by the stack pointer or register r30.

- for vortex, 63% of the ages higher than 100 are caused by the callee-saved registers or registers r9 through r14. According to the calling convention on the Alpha architecture, these registers need to be saved by the callee of a procedure call. As such, the callee will save these registers at the beginning of the procedure (register read) and will restore its state at the end of the procedure (register write). A long time between this register write and a subsequent register read will cause large ages of register instances.
- for m88ksim, 68% of the ages higher than 100 are caused by the argument registers or registers r16 through r21. These registers are used to communicate arguments to a procedure.

For the other benchmarks in the SPECint95 suite, the large ages are caused by a combination of the mechanisms mentioned above.

It is interesting to note that a power law distribution with PDF $P\left[X = x\right] = \alpha x^{-\beta}$ has an infinite mean when $\beta \leq 2$. The slopes of the age, the useful lifetime and the lifetime of register instances PDFs are ≤ 2, e.g., the slope of the age PDF is 1.59 for the IBS traces. As a result, the average age of register instances, also called the *average dependency distance* between operations by some authors [73, 106], is infinite. Since the execution time of a computer program tends to be finite, in practice this average will be finite as well. However, the average dependency distance is meaningless since it is dependent on the number of instructions profiled. For example, we have measured the average dependency distance for gcc for the first 1 million instructions in the trace (result: 141.6), the first 10 million instructions (result: 730.7), the first 100 million instructions (result: 1783.8) and the whole trace (result: 6761.9). In addition, if we take into account that 80% of the dependency distances are smaller than 26 and 90% of the dependency distance are smaller than 66, we can conclude that it makes no sense to talk about the average dependency distance between operations.

4.2 Distribution fitting

As said in the introduction of this chapter, we will use the fact that register traffic characteristics exhibit power law properties to obtain an analytical workload model that will be used in a hybrid analytical-statistical environment to estimate microprocessor performance. To obtain an abstract workload model that is based on a small number of

parameters, we have to approximate measured distribution functions with theoretical distribution functions by estimating parameters, also called *distribution fitting*. In the analytical workload model, the parameters of the theoretical distribution then describe the workload.

Since the statistical simulation methodology presented in the previous chapters is based on the measured age of register operands distribution, we will further concentrate on this distribution. To estimate the theoretical distribution parameters, we take the following approach. The PDF of the age of register instances $P\left[X=x\right]$ can be written as

$$P\left[X=x\right]=P\left[X=x \mid X\geq x\right]\cdot P\left[X\geq x\right],\ x\geq 1 \tag{4.1}$$

where $P\left[X=x \mid X\geq x\right]$ could be defined as the *conditional dependence probability* $1-p_x$ (p_x corresponds to the *conditional independence probability* defined by Dubey, Adams and Flynn [29]); i.e., p_x is the probability that an operation is independent on an operation that comes x operations ahead in the instruction trace given that the operation is independent of the $x-1$ operations ahead of that operation. Equation 4.1 can be rewritten as follows

$$P\left[X=x\right]=(1-p_x)\cdot(1-\sum_{i=1}^{x-1}P\left[X=i\right]). \tag{4.2}$$

Using induction it can be easily verified that $P\left[X=x\right]$ can be written as

$$P\left[X=x\right]=(1-p_x)\cdot\prod_{i=1}^{x-1}p_i,\ x\geq 1. \tag{4.3}$$

Reverse, calculating p_x from the measured $P\left[X=x\right]$ can be done as follows, see equation 4.2:

$$p_x=1-\frac{P\left[X=x\right]}{1-\sum_{i=1}^{x-1}P\left[X=i\right]}. \tag{4.4}$$

Note that any approximation of the conditional independence probability p_x leads to a normalized distribution for the age of register instances. Indeed, summing $P[X=x]$ over all possible values of $x\geq 1$ always yields 1 for any value of p_x, see formula 4.2. For example, assuming a conditional independence probability that is independent of x,

say p, results in the geometric distribution $P[X = x] = (1-p)p^{x-1}$. This approximation was taken by Dubey, Adams and Flynn [29]. Kamin, Adams and Dubey [75] approximated the conditional independence probability p_x by an exponential function

$$p_x \approx 1 - \alpha e^{-\beta x}, \tag{4.5}$$

where α and β are constants that are determined through regression techniques applied to the measured p_x.

In this dissertation, we propose to approximate p_x by a power law function

$$p_x \approx 1 - \alpha x^{-\beta}. \tag{4.6}$$

This is shown for the IBS trace real_gcc in Figure 4.4: the conditional dependence probability distribution, the age of register instances distribution, the cumulative distribution and the corresponding fitted theoretical distributions are presented. The graphs in Figure 4.4 show that the power law approximation is more accurate than the exponential approximation for real_gcc. In Figures 4.5 and 4.6, the age of register instances distributions together with the exponential approximation and the power law approximation are shown for all the SPECint95 benchmarks and all the IBS traces, respectively. These results clearly reveal that in general the power law approximation is more accurate than the exponential distribution proposed by Kamin, Adams and Dubey. However, for some benchmarks, such as li and compress, the power law approximation does not outperform the exponential approximation. We believe that this is due to the fact that these benchmarks have a smaller instruction footprint compared to the other benchmarks, see Tables 2.3 and 2.4, and thus spend most of their time in tight loops resulting in an age of register instances distribution that drops off more quickly than a power law distribution for larger values of x in a log-log diagram.

The distribution fitting was done by minimizing the sum of squared errors between the theoretical distributions and the measured data of the conditional dependence probability. This minimization gives a higher weight to smaller values of x. This choice is motivated by the fact that we want to use this approximation as an abstract workload model for a hybrid analytical-statistical performance modeling technique in which an accurate approximation of the measured data for small values of x is necessary to obtain accurate performance predic-

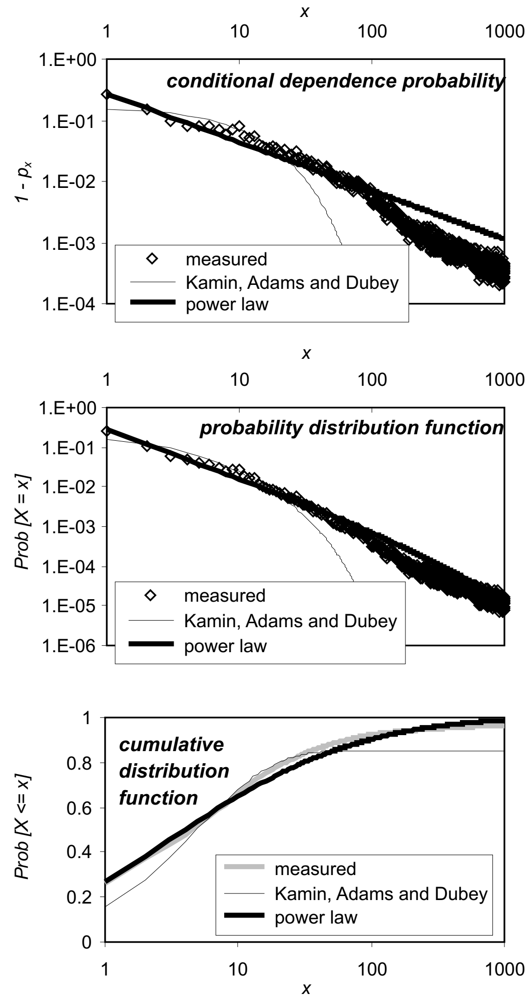


Figure 4.4: Parameter estimation of the conditional dependence probability p_x of the age of register instances for real_gcc. The corresponding probability and cumulative distribution functions are shown as well.

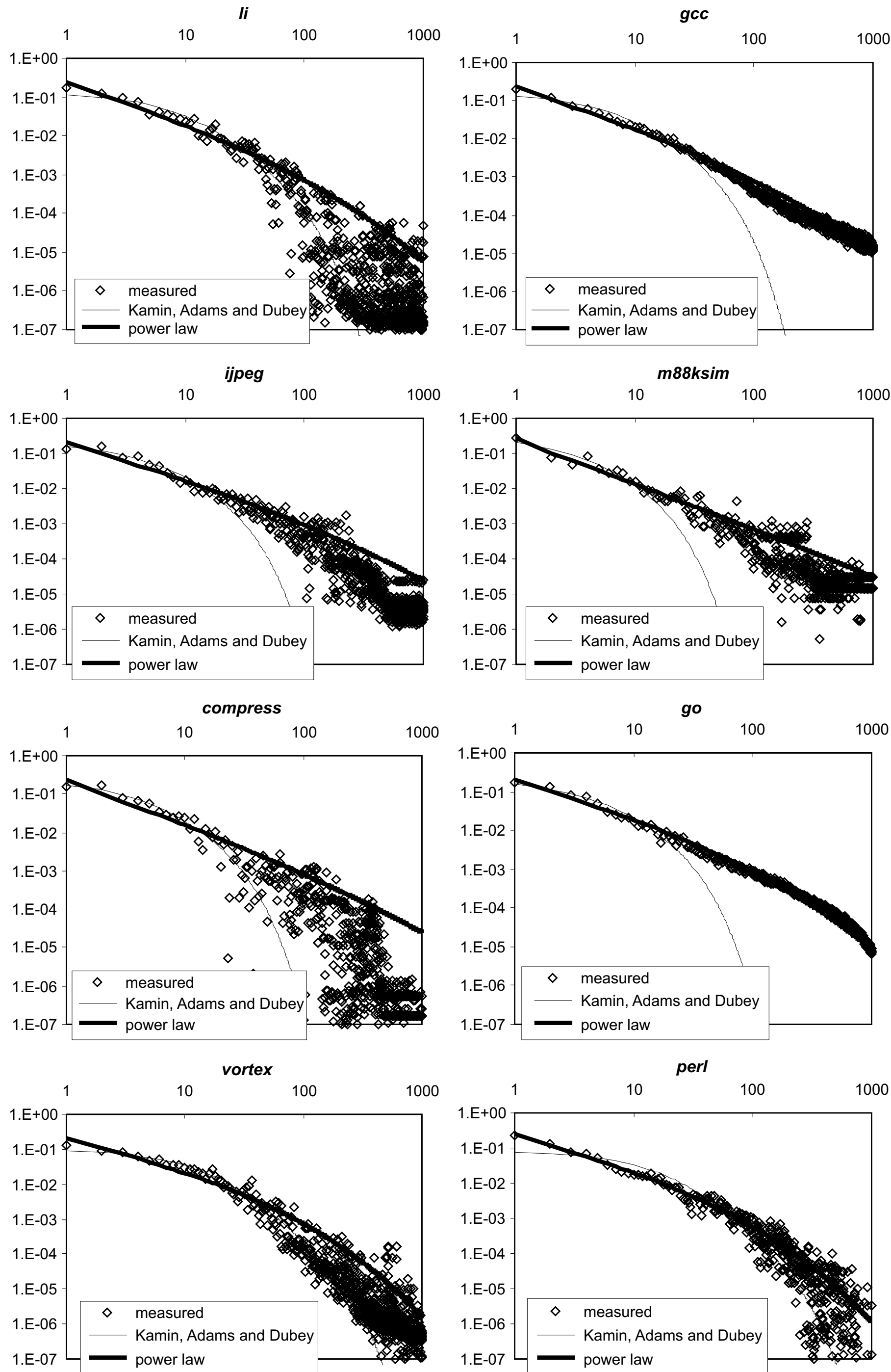


Figure 4.5: Age of register instances for the SPECint95 benchmarks: x on X axis and *Prob [X=x]* on Y axis.

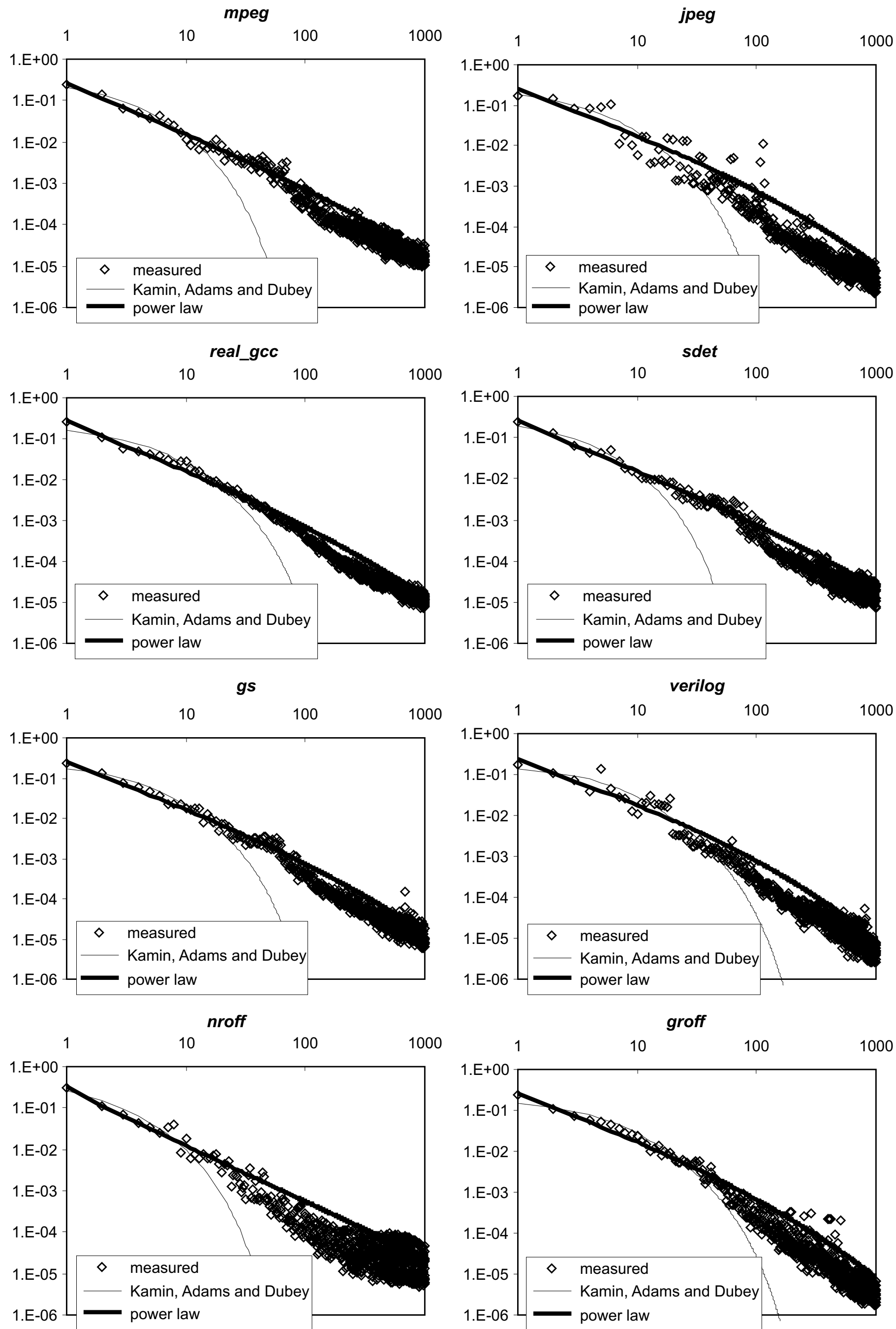


Figure 4.6: Age of register instances for the IBS traces: x on X axis and *Prob [X=x]* on Y axis.

	power law		exponential	
SPECint95	α	β	α	β
li	0.241	0.702	0.120	0.038
gcc	0.234	0.731	0.141	0.068
compress	0.236	0.813	0.200	0.124
go	0.206	0.682	0.172	0.114
ijpeg	0.209	0.765	0.178	0.121
vortex	0.199	0.598	0.093	0.020
m88ksim	0.271	0.922	0.245	0.200
perl	0.240	0.632	0.081	0.019
IBS				
mpeg	0.268	0.825	0.244	0.176
jpeg	0.248	0.749	0.201	0.103
gs	0.259	0.781	0.195	0.116
verilog	0.233	0.714	0.145	0.058
real_gcc	0.270	0.791	0.169	0.088
sdet	0.259	0.835	0.235	0.178
nroff	0.328	0.975	0.323	0.236
groff	0.269	0.751	0.152	0.062

Table 4.1: Fitted α and β values for the power law approximation and the exponential approximation, for the SPECint95 benchmarks and the IBS traces.

tions.

We also experimented with fitting theoretical distributions to the measured PDF data instead of the conditional dependence probability. We found that fitting to the conditional dependence probability generally yields a more accurate approximation, which motivates our approach.

The α and β values obtained from distribution fitting are listed in Table 4.1 for the various benchmarks and for the power law approximation as well as for the exponential approximation. For the power law approximation, α varies between 0.199 and 0.328 and β varies between 0.632 and 0.975. Note that a meaningful interpretation can be given to α and β, namely $\alpha = P[X = 1]$ is the probability that an operation is dependent on its immediately preceding operation and β is the slope of the conditional dependence probability in a log-log diagram.

4.3 Hybrid analytical-statistical modeling

We make use of the information obtained in the previous section to transform the 'classical' statistical simulation technique, as described in the previous chapters, into a methodology that bridges the gap between analytical modeling and statistical simulation. Indeed, we can use the power law distribution and the fitted α and β values as an approximation for the age of register instances distribution. As such, we will end up with an analytical workload model that includes a limited number of workload parameters.

First, we will detail how we transform a statistical profile as discussed in the previous chapter to a statistical profile that will serve as an analytical workload model. Afterwards, the accuracy of the hybrid analytical-statistical modeling technique will be evaluated.

4.3.1 Analytical workload model

Recall that a statistical profile consists of a number of characteristics.

- The *instruction mix* distribution remains unchanged. This distribution contains 19 probabilities representing operation classes according to their types and the number of source operands.
- *The age of register instances* distribution is approximated by the theoretical distributions, i.e., the exponential distribution and the power law distribution, as discussed in the previous section. As such this distribution can be represented by its theoretical parameters, namely α and β. It is also important to note that the measured conditional dependence probability on which the distribution fitting is done, is averaged over all register instances; no distinction is made per instruction type and per operand as is done in the 'classical' statistical simulation technique.
- The *age of memory instances* distribution is not included in the analytical workload model to minimize the number of parameters in the model.
- The *branch statistics* are included which contains 7 probabilities in total.
- The *(simple) cache statistics* are included: 2 probabilities to characterize the I-cache behavior and 2 probabilities to characterize the

D-cache behavior.

In conclusion, the abstract workload model only contains a few parameters: the instruction mix (19 probabilities), α and β to characterize the register traffic and 11 probabilities to characterize the branch and cache behavior.

4.3.2 Evaluation

Absolute accuracy

In Figure 4.7, we have plotted the IPC prediction errors (see formula 2.1; positive error means overestimation) for the various benchmarks and for various configurations of the performance estimation methodology: (i) using the 'classical' statistical simulation technique as discussed in the previous chapter, (ii) using the power law approximation, and (iii) using the exponential approximation proposed by Kamin, Adams and Dubey [29].

The IPC prediction errors for the hybrid analytical-statistical technique are comparable to the 'classical' statistical method: hybrid analytical-statistical modeling is nearly as accurate as 'classical' statistical simulation. When comparing the power law approximation versus the exponential approximation, we can conclude that a better fit of the power law approximation, see section 4.2, does not necessarily lead to a smaller prediction error. There are three reasons explaining this: (i) a better fit for dependency distances larger than the instruction window size has no effect on performance, (ii) the fits are not perfect (although the global shape of one fit can be better than another fit, inaccuracies can occur in individual points of the distributions), and (iii) performance is dependent on the interaction between various characteristics which is not modeled in the statistical simulation methodology—statistical independence is assumed between all characteristics. For gcc for example, the power law approximation is definitely more accurate than the exponential approximation; however, the exponential approximation leads to a comparable performance prediction error as the power law approximation. For perl on the other hand, performance prediction using the power law approximation is more accurate than using the exponential approximation. The reverse is true for li and compress, for which the exponential distribution results in a better fit, see section 4.2. For li, the exponential approximation leads

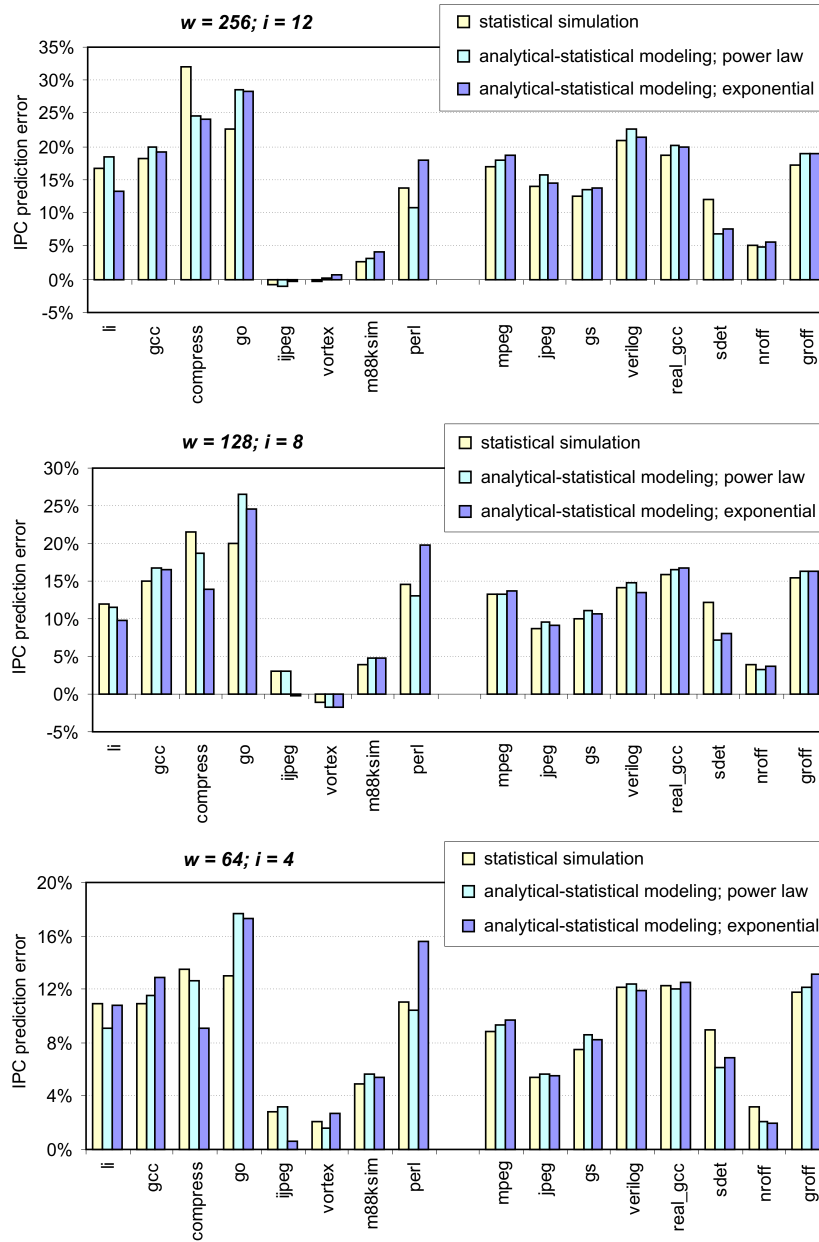


Figure 4.7: Performance prediction accuracy of the statistical and the hybrid analytical-statistical simulation methodology using the power law approximation as well as the exponential approximation for various processor configurations by varying the window size w and issue width i; the 'large' cache configuration was assumed.

to higher performance prediction accuracies; for compress, the power law approximation leads to higher accuracies. Although the power law approximation does not always lead to a more accurate performance prediction, we have to emphasize that the power law approximation is a more accurate model for the register traffic characteristics as is clearly demonstrated in section 4.2.

Relative accuracy

A modeling abstraction can also be evaluated by its relative accuracy, next to its absolute accuracy considered above. Three examples of sensitivity analyses are shown in Figure 4.8:

- the sensitivity to load latency: performance degradation by increasing the load latency (in case of a L1 D-cache hit) from 3 to 4 cycles on a 12-issue 256-entry window processor;
- issue width: performance gain by increasing the issue width from 8 to 16 in a 128-entry window processor; and
- window size: performance gain by increasing the window size from 128 to 512 in a 16-issue processor.

The graphs in Figure 4.8 indicate that the 'classical' statistical and the hybrid analytical-statistical modeling technique are useful to make viable design decisions because the estimated gradients give a good indication of the real gradients in the design space.

4.4 Workload space exploration

Theoretical workload modeling, of which hybrid analytical-statistical modeling is an example, is useful to provide insight in program behavior and its implications on performance. In section 4.2, the conditional dependence probability was fitted to the power law distribution with parameters α and β. Recall that α is the probability that an operation is dependent on its immediately preceding operation and that β is the slope of the conditional dependence probability in a log-log diagram. From this observation we can conclude that α and β are an indication for the amount of ILP (instruction-level parallelism) available in a program. A high β and low α indicate few dependencies over short distances, and thus many dependencies over longer distances and thus

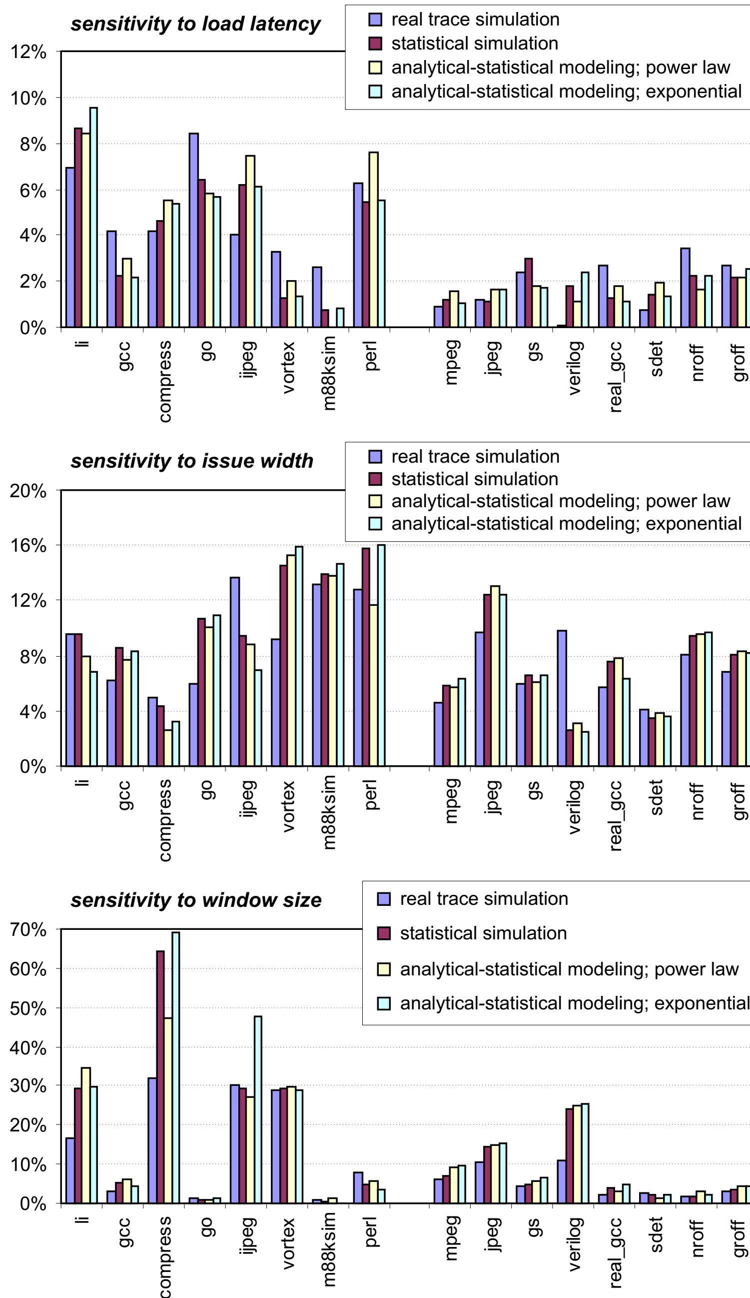


Figure 4.8: Relative accuracy: performance degradation when increasing the load latency from 3 cycles to 4 cycles (upper graph); performance gain when increasing the issue width from 8 to 16 in a 128-entry window processor (graph in the middle); performance gain when increasing the window size from 128 to 512 in a 16-issue processor (lower graph).

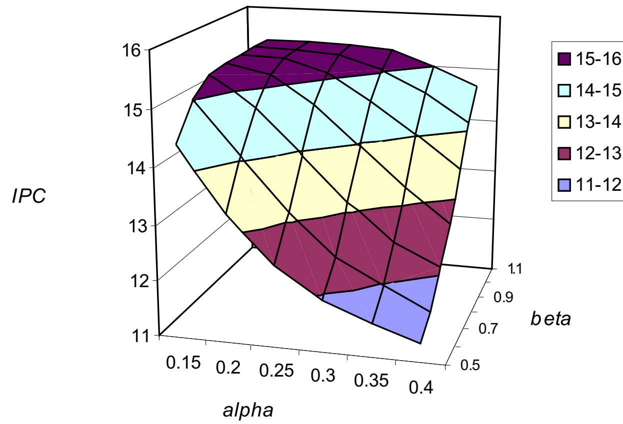


Figure 4.9: Workload space: IPC as function of α and β for a 16-issue, 512-entry window processor with perfect caches and perfect branch prediction.

more ILP. Reversely, a low β and high α indicate many dependencies over shorter distances and thus less ILP. From a compiler point of view, β can be increased and (simultaneously) α can be decreased by techniques such as loop unrolling, scheduling, etc.

An important advantage of hybrid analytical-statistical modeling over 'classical' statistical simulation and real benchmark simulation is that workload space explorations can be done efficiently. Theoretical workload models allow us to explore the *entire* workload space by specifying a limited number of parameters. Real benchmark suites on the other hand, only provide a sample from the entire workload space; and although it is possible to explore the workload space by 'classical' statistical simulation, it is highly impractical due to the large number of probabilities that need to be specified. An example of a workload space exploration is shown in Figure 4.9: IPC as a function of α and β for a 16-issue 512-entry window processor with perfect caches and perfect branch prediction. This figure indeed confirms our reasoning: a high β and a low α result in high ILP; a low β and a high α result in low ILP. The reason why the instruction throughput (IPC) saturates for low values of α and high values of β is that the instruction throughput is limited by the available machine parallelism.

It is also interesting to note that there exist points in the workload

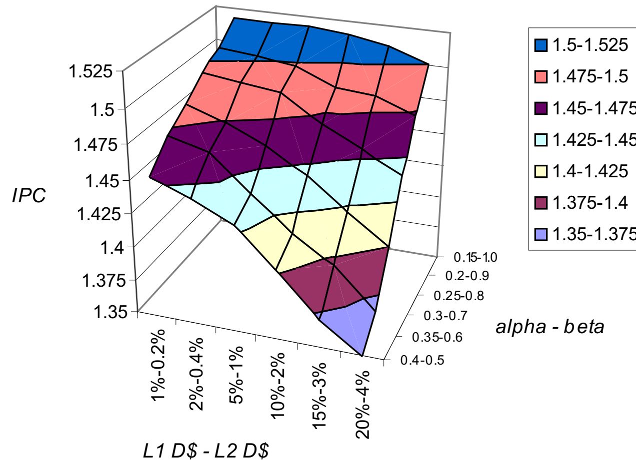


Figure 4.10: Workload space: IPC as function of α, β, L1 and L2 D-cache miss rate for a 16-issue, 512-entry window processor with a 2% L1 I-cache miss rate, a 0.5% L2 I-cache miss rate and a 97% branch prediction accuracy.

space where different values for α and β result in the same performance, e.g., $\alpha = 0.2$ and $\beta = 0.7$ vs. $\alpha = 0.25$ and $\beta = 0.8$.

Theoretical workload modeling can also be used to investigate the interaction between program characteristics that are hard to vary in real programs: e.g., the interaction between the age of register instances and the D-cache miss rate, as is shown in Figure 4.10 for a 16-issue, 512-entry window machine; the L1 I-cache miss rate was set to 2%, the L2 I-cache miss rate was set to 0.5% and the branch prediction accuracy was set to 97%. In this graph, several combinations of L1 and L2 D-cache miss rates are shown on the X axis; several combinations of α and β are shown on the Y axis—$\alpha = 0.4, \beta = 0.5$ suggesting low ILP to $\alpha = 0.15, \beta = 1.0$ suggesting high ILP; the IPC is shown on the vertical Z axis. This graph clearly shows that the performance of a program with high ILP is less affected by the D-cache miss rate than a program with low ILP. This conclusion is taken from the fact that the curve as a function of D-cache miss rate of the $(\alpha = 0.15, \beta = 1.0)$ case is flatter than the curve of the $(\alpha = 0.4, \beta = 0.5)$ case. In other words, the latency of load operations that miss in the data cache can be effectively hidden by a longer age of register instances. By fixing the D-cache miss rate and looking at different performance curves as a function of α and β,

we also observe that the performance improvement by enlarging the age of register instances (for example through compiler optimizations) will be modest for applications with low D-cache miss rates.

The idea of workload space explorations is not new. Similar workload space explorations were done in previous work but all of them used a workload model that does not resemble real program characteristics which may lead to incorrect conclusions. Dubey, Adams and Flynn [29] assumed a constant conditional independence probability p_x which is certainly not the case as is clearly demonstrated in section 4.2; Kamin, Adams and Dubey [75] assumed an exponential approximation of the conditional independence probability which is less accurate than the power law approximation proposed in this dissertation, see section 4.2. Oskin, Chong and Farrens [106] used the average dependency distance between instructions to impose register dependencies which is unrealistic as well, see section 4.1. The results presented in this chapter show that using a power law approximation will yield more realistic workload space explorations.

4.5 Summary

In this chapter we have shown that the register traffic characteristics of a computer program exhibit a power law or a Pareto distribution, i.e., shows a straight line in a log-log diagram. Therefore, we have measured four register traffic distributions: the age, the lifetime, the useful lifetime and the degree of use of register instances. We have fitted these measured distributions to theoretical distributions, namely the previously proposed exponential distribution and the newly proposed power law distribution, and we found that the power law distribution generally yields a better fit. Subsequently, we have used these theoretical parameters in an analytical workload model that summarizes a program behavior in a limited number of parameters. This analytical model was then used in a hybrid analytical-statistical model that achieves an absolute and a relative accuracy that is nearly as good as the 'classical' statistical simulation methodology as presented in the two previous chapters. In addition, we show that such an analytical workload model is extremely useful for exploring the workload space since the various parameters in the model can be varied freely. This allows us to explore the workload space whereas existing benchmark suites only provide a limited number of points in this space.

Chapter 5

Power modeling

Power dissipation is rapidly becoming a limiting factor in high performance microprocessor design due to ever increasing device counts and clock rates.
M. K. Gowan, L. L. Biro and D. B. Jackson in [59]

Companies need to address power consumption issues early in their product development cycles to better determine how best to reduce power consumption during those cycles.
David Cohn, director of IBM's Austin Research Lab

This chapter discusses how statistical simulation can be used in combination with architectural power modeling to estimate power consumption in the earliest stages of the design. We also show that this approach is useful for getting insight in the impact of program characteristics on power consumption.

5.1 Introduction

Power dissipation and energy consumption are and will continue to be key design issues when designing microprocessors. For laptop computers and handheld devices, battery life is the major design constraint. For more high end systems, power consumption is becoming a major design issue as well [59]: higher clock frequencies, larger die sizes and

more and more complex microarchitectures result in a significant increase in power consumption which leads to an increased heating and thus an increased packaging and cooling cost. As a consequence, it is important that power consumption is considered early in the design cycle so that computer engineers are not confronted with an unacceptable power dissipation in a late design phase. Recent work [16, 17, 28, 133, 138, 140] addresses this issue by integrating power modeling techniques in architectural simulators. In other words, power models of various processor structures are combined with counters that measure the activity at the architectural level of each of these structures to calculate the total power consumption of a microprocessor. Unfortunately, because of the fact that these power modeling methodologies are based on detailed architectural simulation, they equally suffer from long simulation times which is exactly the problem that we are trying to solve with statistical simulation.

In this chapter, we address this issue by integrating architectural power modeling in the statistical simulation methodology. This will allow us to efficiently identify a region of interest with desirable power/performance characteristics in the earliest stages of the design cycle. For this purpose, we have integrated an architectural power model, namely Wattch [17], into our trace-driven simulator so that both power and performance characteristics are being measured. We evaluate the accuracy of this approach by comparing power and performance estimates using real and synthetic traces and we find that statistical simulation is indeed capable of identifying a region of energy-efficient architectures. In addition, we show that statistical simulation allows computer designers to investigate the influence and the interaction of several program characteristics that are hard to vary using real programs, such as branch prediction accuracy and cache miss rate, on both performance and energy consumption per cycle.

5.2 Power modeling

5.2.1 General concepts

The total power dissipation of a microprocessor chip can be calculated as follows [13, 54]:

$$P = C \cdot V_{dd} \cdot V_{swing} \cdot a \cdot f + I_{leakage} \cdot V_{dd} + I_{sc} \cdot V_{dd}. \tag{5.1}$$

The first term in this sum represents the switching loss or the dynamic power dissipation, the second term represents the leakage loss and the third term represents the short-circuit loss. In this formula, C is the load capacitance (the internal capacitances of the circuits that make up the processor; this includes transistors as well as wires), V_{dd} is the supply voltage (typically 1V to 3V), V_{swing} is the maximum voltage swing across the load capacitance, a is the activity factor ($0 < a \leq 1$), which measures how often clock ticks lead to switching activity on average, f is the clock frequency, $I_{leakage}$ is the leakage current and I_{sc} is the short-circuit current. In the literature, V_{swing} is often approximated by V_{dd}, V for short. According to [54], the switching loss component remains the dominant term in equation 5.1. As such, the total power consumption of a microprocessor chip can be approximated by its dynamic power dissipation:

$$P = C \cdot V^2 \cdot a \cdot f. \tag{5.2}$$

This approximation forms the basis for architectural power modeling.

5.2.2 Architectural power modeling

Several architectural-level power estimation models have been proposed in the last few years, e.g., Wattch [17], SimplePower [133], PowerTimer [16] and TEM^{2}P^{2}EST [28]. In this study, we use Wattch as the power estimation model because of its public availability and its flexibility for architectural design space explorations; the power models included in Wattch are fully parameterizable which provides its high flexibility. Wattch also provides good *relative* accuracy which is required for doing architectural design space explorations, see previous chapters.

Wattch [17] calculates the dynamic power dissipation P of a processor unit (e.g., functional unit, I-cache, D-cache, register file, clock distribution, etc.) using formula 5.2. The total dynamic power dissipation of the processor is then calculated as:

$$\begin{aligned} P &= \sum_{i=1}^{N} C_i \cdot V_i^2 \cdot a_i \cdot f_i \\ &= V^2 \cdot f \cdot \sum_{i=1}^{N} C_i \cdot a_i, \end{aligned} \tag{5.3}$$

with C_i, V_i, a_i and f_i the load capacitance, the supply voltage, the activ-

ity factor and the frequency for each processor unit, respectively; since we assume that the voltage and the frequency is identical over the complete chip, V and f can be placed outside the sum in the equation. In this study, we assume $V = 2.5$V and a .35μm technology. The clock frequency f is unused in this study since the data presented in this chapter is energy per cycle. The reason for this choice is that Wattch was validated against reported data for these technology parameters. Wattch estimates the capacitance C_i for each processor unit based on circuit and transistor sizing models. The activity factors a_i measure how often clock ticks lead to switching activity on average for each processor unit. In Wattch, a_i is measured by looking at the data being generated at various points in the microarchitecture during the architectural simulation and by measuring how these data lead to switching activity. In this study however, we assume a base activity factor of $1/2$ modeling random switching activity, which we believe is a reasonable approximation in an early design stage[1]. The activity factors a_i can be further lowered by clock-gating unneeded units. In this study, we assume an aggressive conditional clocking scheme in which the power estimate is linearly scaled from the maximum power dissipation with port or unit usage; unused units dissipate 10% of their maximum power. Measuring the unit usage is done by inserting so called *activity counters* in the simulator that keep track of the number of accesses per clock cycle to the various units.

For this study, we have integrated Wattch in our trace-driven simulator by inserting the Wattch activity counters in our simulator. The architecture modeled is based on an instruction window that serves as both reorder buffer (ROB) and issue window, see section 2.1, as is done with the register update unit (RUU) design of SimpleScalar's `sim-outorder` [18] on which Wattch is built. We also assume 3 extra pipeline stages between the fetch and the issue stage, as is done in Wattch. In addition, our simulator also models a load/store queue that is used in our case for dynamic memory disambiguation. Ghiasi and Grunwald [57] evaluated two architectural power models and concluded that the architecture modeled in the simulator should be the same as the one used by the power model to produce reliable results. The above discussion shows that this is the case in this study.

[1] An alternative approach would be to use a distribution of the data values produced in a program execution for generating synthetic data values in the synthetic trace. These synthetically generated data values could then be used to measure the switching activity in various structures.

In conclusion, the only difference between the statistical simulation methodology as discussed in the previous chapters and the idea proposed in this chapter, namely statistical simulation for power/performance modeling, is that the trace-driven simulator now also incorporates an architectural power model which yields power metrics next to performance metrics.

5.2.3 Power/performance metrics

To evaluate this methodology, we use several metrics next to the *IPC prediction error* that was considered in previous chapters. Analogously to the IPC prediction error, we measure the *energy per cycle prediction error* which is the relative error between the average energy consumption per cycle of the real trace and its synthetic 'clone' trace; a positive error means an overestimation. Based on IPC predictions and energy per cycle predictions, we can derive predictions for two power/performance metrics, namely the energy-delay product and the energy-delay-square product. These two metrics were found to be reasonable metrics for evaluating the *energy efficiency* of midrange and high-end microprocessor designs [13]. The *energy-delay product* (EDP) is defined as follows:

$$\begin{aligned} EDP &= \frac{energy}{instruction} \cdot \frac{cycles}{instruction} \\ &= \left(\frac{1}{IPC}\right)^2 \cdot \frac{energy}{cycle} \end{aligned} \tag{5.4}$$

and the *energy-delay-square product* (ED^{2}P) is defined as follows:

$$\begin{aligned} ED^2P &= \frac{energy}{instruction} \cdot \left(\frac{cycles}{instruction}\right)^2 \\ &= \left(\frac{1}{IPC}\right)^3 \cdot \frac{energy}{cycle}. \end{aligned} \tag{5.5}$$

Note that these two metrics are fused metrics that measure power and performance together. In [13], Brooks *et al.* argue that the EDP is appropriate for high-end systems, for example workstations. The ED^{2}P is found to be appropriate for the highest performance server-class machines, since the ED^{2}P gives an even higher weight to performance. The inverses of the EDP and the ED^{2}P are proportional to the so called *MIPS*^{2}*/Watt* and *MIPS*^{3}*/Watt* metrics, respectively. In these metrics,

MIPS stands for millions of instructions executed per second and *Watt* for the energy consumed per second.

For completeness and for clarity, we now discuss other power related metrics that are used in the literature and that might be relevant for specific target domains:

- The total amount of energy consumed during a program execution is another important metric. The *energy consumption* E can be calculated as follows:
$$E = P \cdot T, \tag{5.6}$$
with T the execution time of the computer program. In other words, the total energy consumption of a program execution is proportional to the average power dissipation P. Note that energy consumption is not only important for battery-powered devices; for higher end systems that are connected to the electricity grid, energy consumption is directly related to the electricity bill.

- Another metric that might be appropriate for lower end systems is *energy per instruction*, calculated as follows:
$$\textit{energy per instruction} = \frac{1}{IPC} \cdot \frac{energy}{cycle}. \tag{5.7}$$
This metric can also be estimated given an IPC prediction and an energy per cycle prediction. The inverse of this metric is proportional to the *MIPS/Watt* metric.

- Two metrics that are more related to heating, packaging cost, cooling cost and reliability are *maximum power dissipation* and *power density* or power dissipation per unit of area.

5.3 Evaluation

5.3.1 Absolute accuracy

In this section, we evaluate the absolute accuracy of this newly proposed methodology. Figures 5.1 and 5.2 present IPC and energy per cycle obtained by real trace simulation (labeled 'real') and statistical simulation (labeled 'estimated'), measured along the left Y axis. The corresponding prediction errors are measured along the right Y axis.

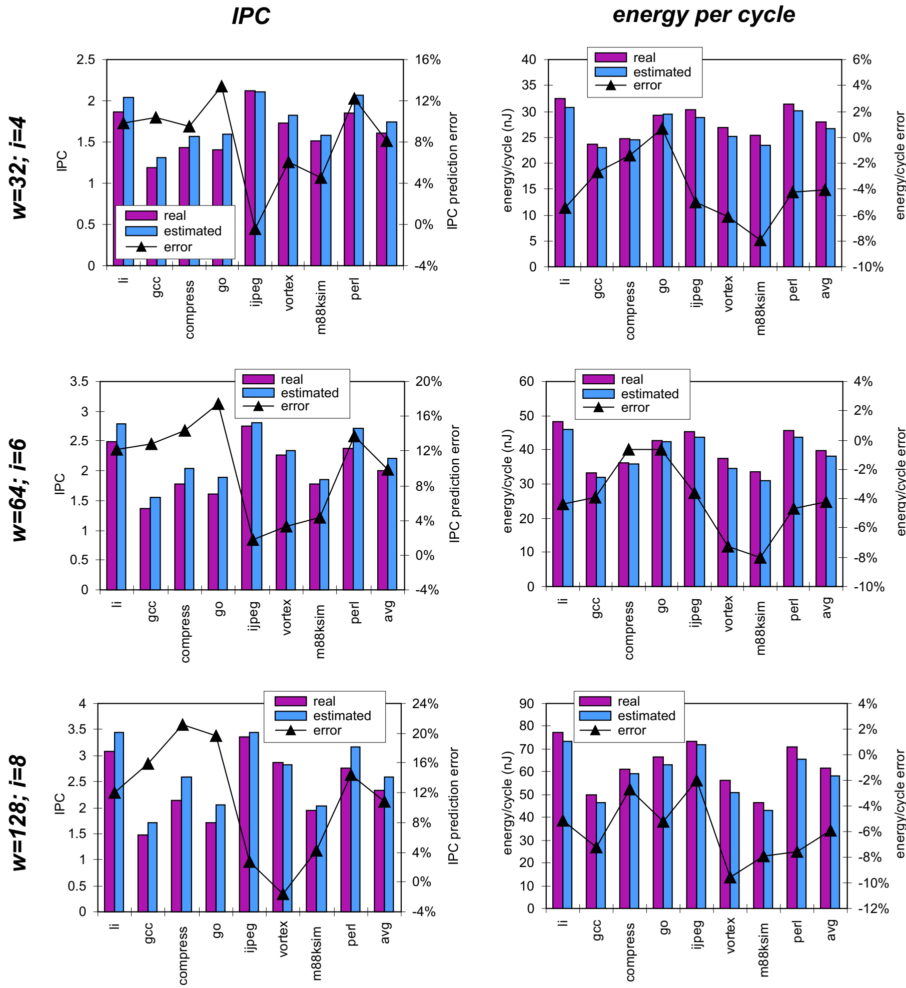


Figure 5.1: Real and estimated IPC (on the left), energy per cycle (on the right) and their corresponding prediction errors for three microarchitectural configurations for the SPECint95 traces.

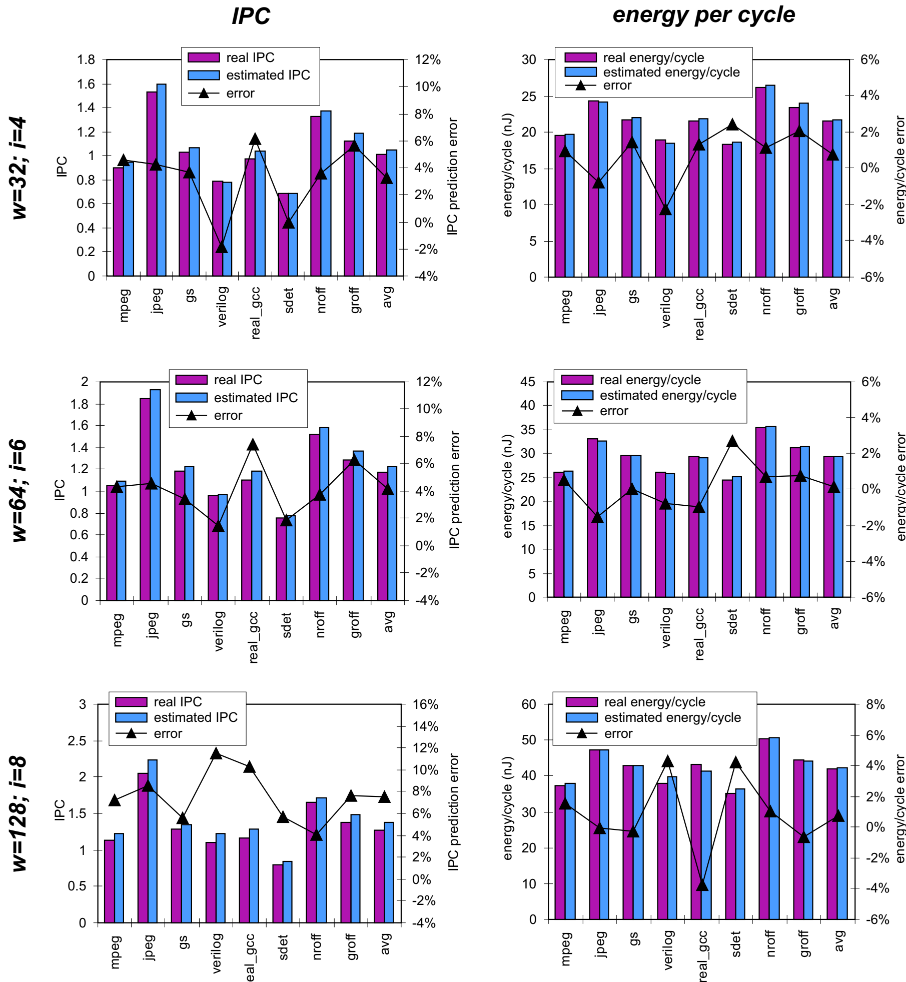


Figure 5.2: Real and estimated IPC (on the left), energy per cycle (on the right) and their corresponding prediction errors for three microarchitectural configurations for the IBS traces.

In addition, the results are shown for three microarchitectural configurations by varying the window size w and issue width i: a 4-issue 32-entry window processor (top graph), a 6-issue 64-entry window processor (middle graph) and an 8-issue 128-entry window processor (bottom graph).

The IPC prediction errors are the same as the ones presented in previous chapters and are no larger than 21% for the SPECint95 traces and 12% for the IBS traces. The energy per cycle prediction errors are no larger than 5%. As pointed out in previous chapters, the IPC prediction error increases for wider resource machines: e.g., for the IBS traces, a maximum IPC prediction error of 6%, 8% and 12% for the 32/4, the 64/6 and the 128/8 configuration, respectively. The same is true for the energy per cycle prediction error: 2%, 3% and 5% for the 32/4 , the 64/6 and 128/8 configuration, respectively.

The reason why the energy per cycle prediction error is smaller than the performance prediction error can be understood intuitively. Power consumption of an individual instruction consists of two components: the basic power consumption per instruction (dependent on the instruction type) and another component that is related to the interaction between instructions. Indeed, every instruction involves an inherent energy consumption irrespective of the overall performance: fetching the I-cache, accessing the branch predictor, accessing the register renaming table, shifting through the pipeline, reading register operands, executing on a functional unit, accessing the D-cache if appropriate, etc. The energy per cycle prediction error thus comes from performance related issues: e.g., long waiting time for dependencies to be cleared, pressure on the register file, functional unit usage, etc.

5.3.2 Relative accuracy

As is extensively argued before, relative accuracy is more important for early stage methodologies than absolute accuracy. Therefore, in this section we evaluate the relative accuracy of statistical simulation for early design stage power modeling.

Figure 5.3 presents the 'real' (obtained through real trace simulation) and the 'estimated' (obtained through statistical simulation) energy per cycle as a function of issue width for a 128-entry window processor (in the top graphs) and as a function of window size for an 8-issue processor (in the bottom graphs). These data are averaged over all

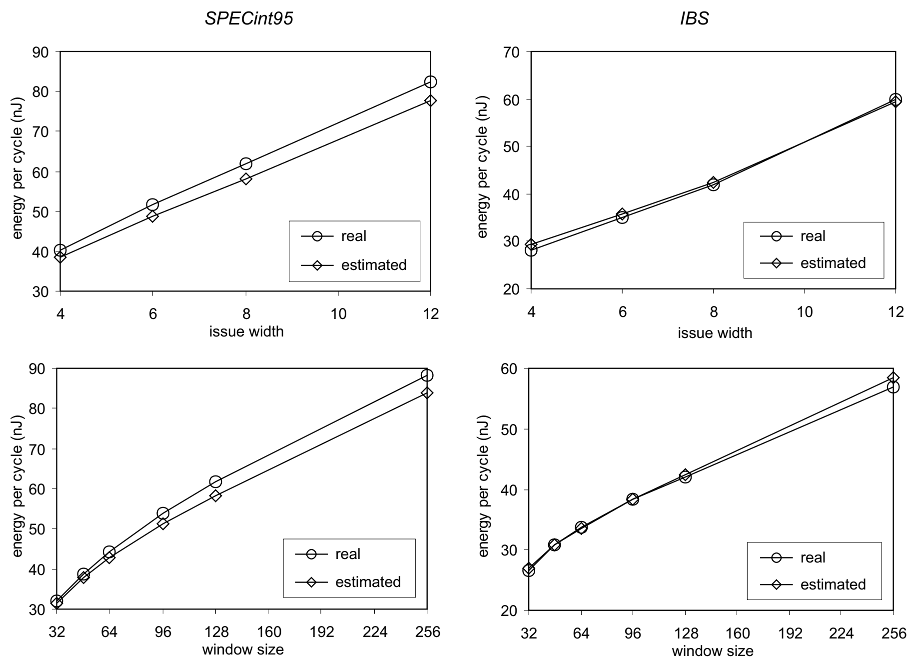


Figure 5.3: Real and estimated energy per cycle as a function of issue width for a 128-entry window processor (top) and as a function of window size for an 8-issue processor (bottom).

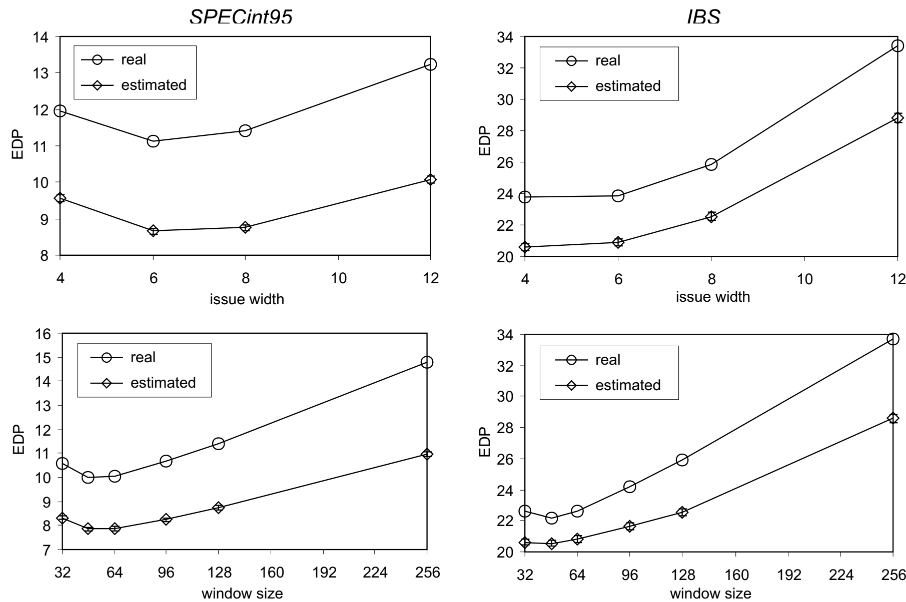


Figure 5.4: Real and estimated energy-delay product (EDP) as a function of issue width for a 128-entry window processor (top) and as a function of window size for an 8-issue processor (bottom).

the SPECint95 traces and the IBS traces, respectively. Figure 5.3 clearly shows that statistical simulation attains excellent relative accuracy for predicting the energy consumption per cycle.

5.3.3 Energy efficiency

The purpose of this section is to verify whether statistical simulation is accurate enough for identifying energy efficient microarchitectures. As discussed in section 5.2.3, we use two power/performance metrics to measure the energy efficiency of a given microarchitecture, namely the energy-delay product (EDP) and the energy-delay-square product (ED^2P). The EDP and the ED^2P are shown in Figures 5.4 and 5.5, respectively. This is done as a function of the issue width for a 128-entry window microarchitecture in the upper graph of each figure and as a function of the window size for an 8-wide microprocessor in the bottom graph of each figure. The jitter (1% for the EDP and 1.8% for the ED^2P) is shown as well for the power/performance metrics ob-

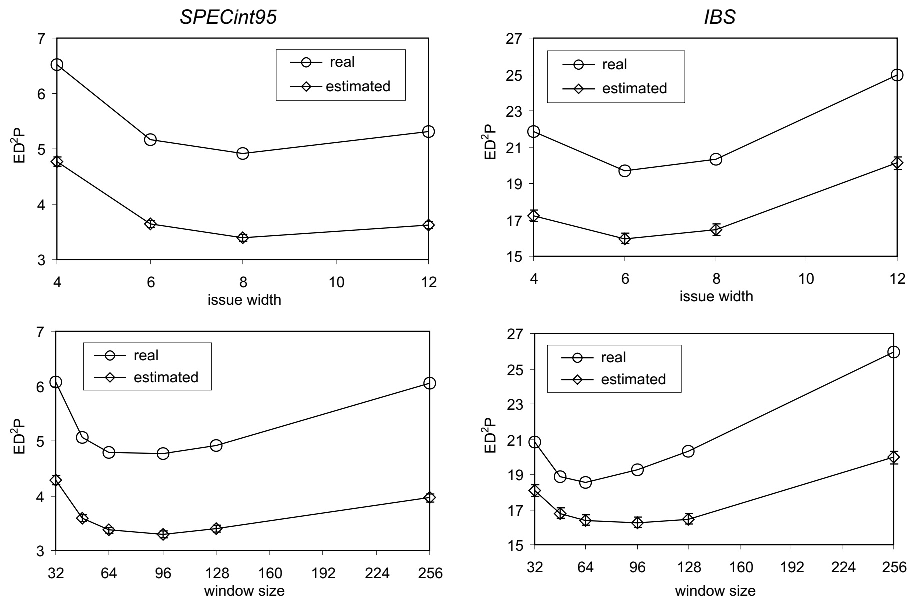


Figure 5.5: Real and estimated energy-delay-square product (ED^{2}P) as a function of issue width for a 128-entry window processor (top) and as a function of window size for an 8-issue processor (bottom).

tained through statistical simulation, labeled 'estimated'. From these figures, we can conclude that although statistical simulation is not always capable of identifying *the* most energy efficient microarchitecture, a region of interesting design points is identified adequately, see for example in the bottom graph of Figure 5.5. For the IBS traces, real trace simulation identifies the 64-entry window machine as the most energy efficient microarchitecture; statistical simulation on the other hand predicts that the microarchitectures with a window size varying between 48 and 128 are the most energy-efficient. There are two reasons for this phenomenon. First, the absolute and the relative errors are different for the IPC predictions and the energy per cycle predictions resulting in additional errors when both metrics are combined in the power/performance metrics. Second, the jitter on the power/performance predictions (shown by a vertical line in Figures 5.4 and 5.5) leads to an uncertainty making it sometimes difficult to identify the optimal configuration. The definition of the jitter on EDP and ED^2P will be given in the next section.

5.4 Energy behavior vs. program characteristics

Note that statistical simulation is maybe not the most suitable tool for exploring the most energy-efficient branch predictor or memory hierarchy. This is due to the distinction made between microarchitecture-dependent and microarchitecture-independent characteristics, see chapter 2. To evaluate two different memory hierarchies for example, cache miss rates have to be computed by simulating the real trace for both hierarchies. In this case, full blown architectural simulations using real benchmarks (simulating the architecture and the memory hierarchy simultaneously), although slower than simulating a memory hierarchy and subsequently doing a statistical simulation, are likely to be more appropriate due to their higher accuracy.

On the other hand, statistical simulation seems to be more suitable for investigating the energy consumption *per cycle* and its interaction with program characteristics. This addresses the question at what time during the execution of a computer program, energy is consumed due to which program characteristic. This is an important issue for techniques such as dynamic thermal management [14] and dynamic adaptation of hardware resources [19]. These techniques search at reducing the energy consumption per cycle (e.g., to control the temperature

of the chip) without compromising performance too much. Statistical simulation combined with power modeling gives an excellent opportunity to investigate this relation since program characteristics can be varied freely and independently.

Figure 5.6 explores IPC and energy per cycle as a function of I- and D-cache miss rate for a 6-issue architecture with a 128-entry window. The branch prediction accuracy was set to 95%—the graphs for other branch prediction accuracies are very similar. We did these experiments because it is unclear in advance how energy per cycle will vary with cache miss rate. On the one hand, instruction throughput will be lower with an increasing cache miss rate. Thus, we might expect an energy per cycle reduction. On the other hand, a higher cache miss rate results in more accesses to higher levels in the memory hierarchy[2] which might lead to higher energy consumptions since larger memory structures consume more energy per access than smaller memory structures. Therefore, it is unclear if higher cache miss rates will result in a higher or lower energy consumption per cycle. The data of Figure 5.6 reveal that generally energy consumption per cycle decreases with increasing miss rates. For low levels of ILP (or high I-cache miss rates) however, the energy per cycle slightly increases (not visible on the graph) with increasing D-cache miss rates. From these experiments, we can conclude that the impact of the reduction in pipeline throughput on the energy consumption per cycle is more significant than the increase in activity per cycle in the memory hierarchy with increasing cache miss rates. Note again that this conclusion is only true as far as the on-chip energy consumption is concerned since (off-chip) memory power consumption is not modeled in Wattch.

In the next set of experiments, we focus on the influence of branch mispredictions on the energy consumption per cycle. Figure 5.7 quantifies the impact of branch prediction accuracy and cache miss rates on IPC and energy per cycle. Several configurations were considered with varying cache miss rates, see the legend of Figure 5.7. The different points on each line correspond to different branch prediction accuracies: 80%, 90%, 95%, 98% and 100% from left to right, respectively. This figure reveals an interesting result, namely that energy per cycle is positively correlated with branch prediction accuracy for low cache miss rates. For high cache miss rates on the other hand, energy per cycle is

[2]Note that only on-chip memories, i.e., L1 and L2 caches, are modeled in Wattch; energy consumption due to main memory accesses is not considered here.

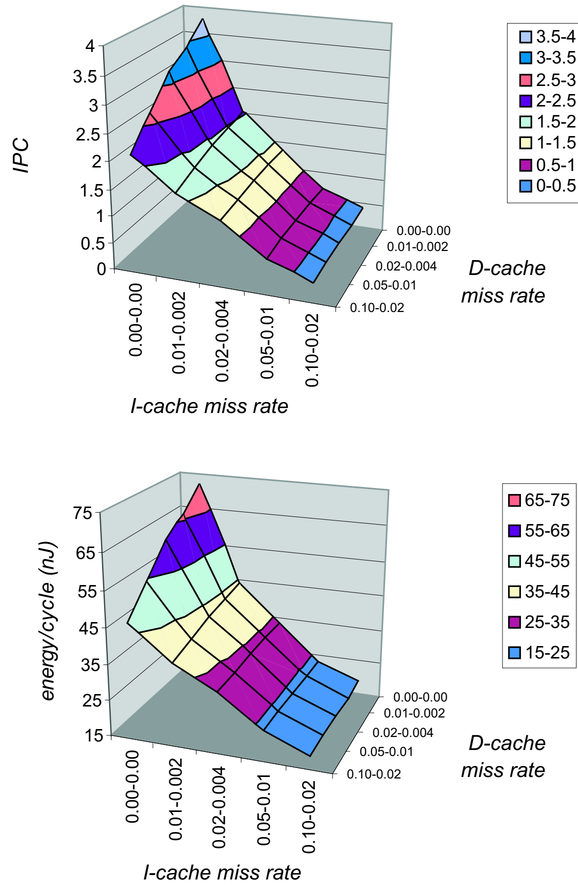


Figure 5.6: IPC (top graph) and energy consumption per cycle (bottom graph) as a function of I- and D-cache miss rate for a 6-issue architecture with a 128-entry window. The branch prediction accuracy was set to 95%. Note that '0.02-0.004' means a 2% L1 miss rate and a 0.4% global L2 miss rate.

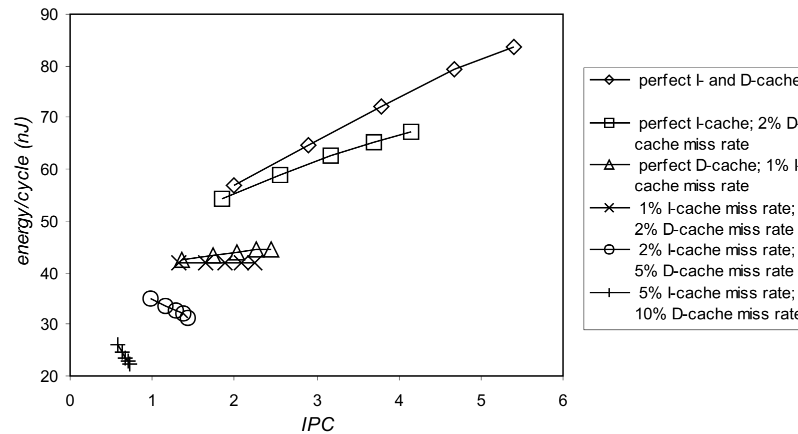


Figure 5.7: IPC and energy per cycle as a function of branch prediction accuracy for a 6-issue architecture with a 128-entry window. The different points on each line correspond to the branch prediction accuracy: 80%, 90%, 95%, 98% and 100% from left to right, respectively.

negatively correlated with branch prediction accuracy. This can be explained as follows, see Figure 5.8. When a branch misprediction occurs, a significant part of the instruction window needs to be squashed and refilled with instructions from the correct control flow path. While refilling the instruction window, the fetch activity is increased compared to the steady state situation without branch mispredictions. The instruction execution activity on the other hand is decreased since fewer instructions reside in the instruction window. As a result, there is an increased activity in the front end of the pipeline and a decreased activity in the back end while recovering from a branch misprediction. In the case of high cache miss rates (and thus low average IPC), see Figure 5.8 (top), the increase in activity in the front end is larger than the decrease in the back end. As a net result, the energy consumption per cycle will be larger while recovering from a branch misprediction. So, the energy consumption per cycle increases with an increasing number of branch mispredictions. In other words, energy consumption per cycle decreases with higher branch prediction accuracies, see Figure 5.7. In case of low cache miss rates (and thus high average IPC) on the other hand, see Figure 5.8 (bottom), the increase in activity in the front end is smaller than the decrease in the back end with a net decrease as result.

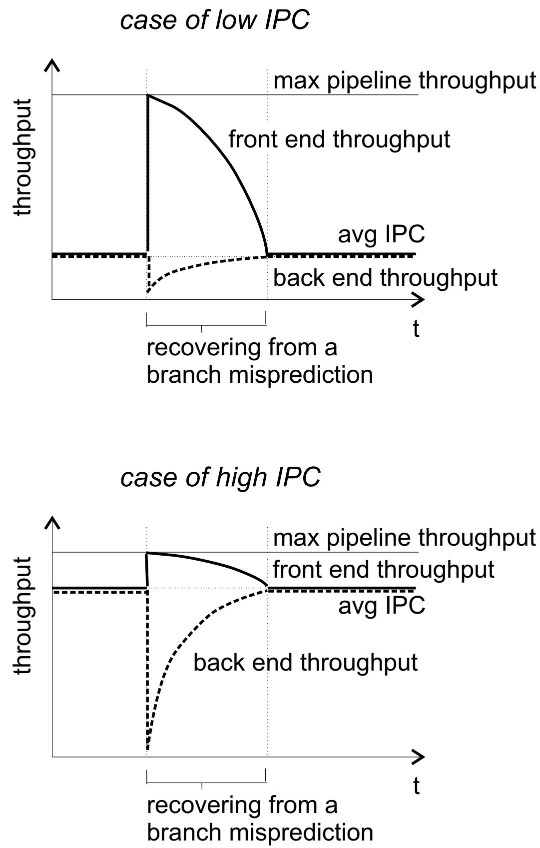


Figure 5.8: Front end and back end activity on a branch misprediction in case of low IPC (top graph) and high IPC (bottom graph).

So, the energy consumption per cycle increases with higher branch prediction accuracies, see Figure 5.7.

Next to providing insight in the power/performance behavior of computer programs, this kind of experiments could be useful for identifying new microarchitectural techniques that reduce the energy per cycle consumption without sacrificing on performance. For example, see Figure 5.7, increasing the branch prediction accuracy in a program with high average cache miss rates leads to an IPC increase and an energy per cycle decrease.

5.5 Summary

Power consumption is becoming a key design issue even for high-performance microprocessor designs. As such, it should be considered in the earliest stages of the design cycle so that computer engineers are not confronted with unexpected power dissipations and heating in the latest stages of the cycle. Such late design surprises might lead to a redesign or to unexpected additional costs for packaging or cooling which reduces the margin of profit. To address this issue, several research groups have proposed to integrate power models in architectural simulators so that power can be considered together with performance while doing microarchitectural design space explorations. However, these simulations suffer from huge simulation times. As such, we propose to combine architectural power modeling with statistical simulation which was evaluated in this chapter. Our results indicate that statistical simulation attains excellent absolute and relative accuracy when it comes to energy consumption per cycle. We also found that this methodology can also be used to identify a region of energy efficient microarchitectures. Finally, we show that this methodology is useful for investigating the interaction between program characteristics and energy consumption per cycle.

Chapter 6

Workload design

Statistics are like a bikini. What they reveal is suggestive,
but what they conceal is vital.
Aaron Levenstein

Having a representative workload of the target domain of a microprocessor is extremely important throughout its design. The composition of a workload involves two issues: (i) which benchmarks to select and (ii) which input data sets to select per benchmark. Unfortunately, we are unable to select a huge number of benchmarks and respective input sets due to limitations on the available simulation time. In this chapter, we use statistical data analysis techniques such as principal components analysis (PCA) and cluster analysis to efficiently explore the workload space. Within this workload space, different input data sets for a given benchmark can be displayed, a distance can be measured between program-input pairs that gives us an idea about their mutual behavioral differences and representative input data sets can be selected for the given benchmark. This methodology is validated by showing that program-input pairs that are close to each other in this workload space indeed exhibit similar behavior. The final goal is to select a limited set of representative benchmark-input pairs that span the complete workload space. Next to workload composition, there are a number of other possible applications, namely getting insight in the impact of input data sets on program behavior and profile-guided compiler optimizations.

6.1 Introduction

The first step when designing a new microprocessor is to compose a workload that should be representative for the set of applications that will be run on the microprocessor once it will be used in a commercial product [11, 73]. A workload then typically consists of a number of benchmarks with respective input data sets taken from various benchmarks suites, such as SPEC, TPC, MediaBench, etc. This workload will then be used during the various simulation runs to perform design space explorations. It is obvious that *workload design*, or the composition of a representative workload, is extremely important in order to obtain a microprocessor design that is optimal for the target environment of operation. The question when composing a representative workload is thus twofold: (i) which benchmarks and (ii) which input data sets to select. In addition, we have to take into account that even high-level architectural simulations are extremely time-consuming. As such, the total simulation time should be limited as much as possible to limit the time-to-market. This implies that the total number of benchmarks and input data sets should be limited without compromising the final design. Ideally, we would like to have a limited set of benchmark-input pairs spanning the complete workload space, which contains a variety of the most important types of program behavior.

Conceptually, the complete workload design space can be viewed as a p-dimensional space with p the number of important program characteristics that affect performance, e.g., branch prediction accuracy, cache miss rates, instruction-level parallelism, etc. Obviously, p will be too large to display the workload design space understandably. In addition, correlation exists between these variables which reduces the ability to understand what program characteristics are fundamental to make the diversity in the workload space. In this chapter, we reduce the p-dimensional workload space to a q-dimensional space with $q \ll p$ ($q = 2$ or $q = 3$ typically) making the visualisation of the workload space possible without losing important information. This is achieved by using statistical data reduction techniques such as principal components analysis (PCA) and cluster analysis.

Each benchmark-input pair is a point in this (reduced) q-dimensional space obtained after PCA. We can expect that different benchmarks will be 'far away' from each other while different input data sets for a single benchmark will be clustered together. This representation gives us an excellent opportunity to measure the impact of input data sets on

program behavior. Weak clustering (for various inputs and a single benchmark) indicates that the input set has a large impact on program behavior; strong clustering on the other hand, indicates a small impact. This claim is validated by showing that program-input pairs that are close to each other in the workload space indeed exhibit similar behavior. I.e., 'close' program-input pairs react in similar ways when changes are made to the architecture.

In addition, this representation gives us an idea which input sets should be selected when composing a workload. Strong clustering suggests that a single or only a few input sets could be selected to be representative for the cluster. This will reduce the total simulation time significantly for two reasons: (i) the total number of benchmark-input pairs is reduced; and (ii) we can select the benchmark-input pair with the smallest dynamic instruction count among all the pairs in the cluster.

Another important application, next to getting insight in program behavior and workload composition, is profile-driven compiler optimizations. During profile-guided optimizations, the compiler uses information from previous program runs (obtained through profiling) to guide compiler optimizations. Obviously, for effective optimizations, the input set used for obtaining this profiling information should be representative for a large set of possible input sets. The methodology proposed in this chapter can be useful in this respect because input sets that are close to each other in the workload space will have similar behavior.

This chapter is organized as follows. In section 6.2, the program characteristics involved are enumerated. Principal components analysis, cluster analysis and their use in this context are discussed in section 6.3. In section 6.4, it is shown that these data reduction techniques are useful in the context of workload characterization. In addition, we discuss how input data sets affect program behavior. Section 6.5 discusses related work. We conclude in section 6.6.

6.2 Workload characterization

It is important to select program characteristics that affect performance for performing data analysis techniques in the context of workload characterization. Selecting program characteristics that do not affect performance, such as the dynamic instruction count, might dis-

criminate benchmark-input pairs on a characteristic that does not affect performance, yielding no information about the behavior of the benchmark-input pair when executed on a microprocessor. On the other hand, it is important to incorporate as many program characteristics as possible so that the analysis done on it will be predictive. I.e., we want strongly clustered program-input pairs to behave similarly so that a single program-input pair can be chosen as a representative of the cluster. The determination of what program characteristics to be included in the analysis in order to obtain a predictive analysis is a study on its own and is out of the scope of this chapter. The goal of this chapter is to show that a data analysis techniques such as PCA and cluster analysis can be helpful for getting insight in the workload space when composing a representative workload.

We have identified the following program characteristics:

- **Instruction mix.** We consider five instruction classes: integer arithmetic operations, logical operations, shift and byte manipulation operations, load/store operations and control operations.

- **Branch prediction accuracy.** We consider the branch prediction accuracy of three branch predictors: a bimodal branch predictor, a gshare branch predictor and a hybrid branch predictor. The bimodal branch predictor consists of an 8K-entry table containing 2-bit saturating counters which is indexed by the program counter of the branch. The gshare branch predictor is an 8K-entry table with 2-bit saturating counters indexed by the program counter xor-ed with the taken/not-taken branch history of 12 past branches. The hybrid branch predictor [90] combines the bimodal and the gshare predictor by choosing among them dynamically. This is done using a meta predictor that is indexed by the branch address and contains 8K 2-bit saturating counters.

- **Data cache miss rates.** Data cache miss rates were measured for five different cache configurations: an 8KB and a 16KB direct mapped cache, a 32KB and a 64KB two-way set-associative cache and a 128KB four-way set-associative cache. The block size was set to 32 bytes.

- **Instruction cache miss rates.** Instruction cache miss rates were measured for the same cache configurations mentioned for the data cache.

- **Sequential flow breaks.** We have also measured the number of instructions between two sequential flow breaks or, in other words, the number of instructions between two taken branches. Note that this metric is higher than the basic block size because some basic blocks 'fall through' to the next basic block.

- **Instruction-level parallelism.** To measure the amount of ILP in a program, we consider an infinite-resource machine, i.e., infinite number of functional units, perfect caches, perfect branch prediction, etc. In addition, we schedule instructions as soon as possible assuming unit execution instruction latency. The only dependencies considered between instructions are read-after-write (RAW) dependencies through registers as well as through memory, as is done in [2].

For this study, there are $p = 20$ program characteristics in total on which the analyses are done.

6.3 Data analysis

In the first two subsections of this section, we will discuss two data analysis techniques, namely principal components analysis (PCA) and cluster analysis. In the last subsection, we will detail how we used these techniques for analyzing the workload space in this study.

6.3.1 Principal components analysis

Principal components analysis (PCA) [87] is a statistical data analysis technique that presents a different view on the measured data. It builds on the assumption that many variables (in our case, program characteristics) are correlated and hence, they measure the same or similar properties of the program-input pairs. PCA computes new variables, called *principal components*, which are *linear combinations* of the original variables, such that all principal components are uncorrelated. PCA transforms the p variables $X_1, X_2, \ldots, X_p$ into p principal components $Z_1, Z_2, \ldots, Z_p$ with $Z_i = \sum_{j=1}^{p} a_{ij} X_j$. This transformation has the following properties

1. $Var[Z_1] > Var[Z_2] > \ldots > Var[Z_p]$ which means that Z_1 contains the most information and Z_p the least; and

2. $Cov[Z_i, Z_j] = 0, \forall i \neq j$ which means that there is no information overlap between the principal components.

Note that the total variance in the data remains the same before and after the transformation, namely $\sum_{i=1}^{p} Var[X_i] = \sum_{i=1}^{p} Var[Z_i]$.

As stated in the first property in the previous paragraph, some of the principal components will have a high variance while others will have a small variance. By removing the components with the lowest variance from the analysis, we can reduce the number of program characteristics while controlling the amount of information that is thrown away. We retain q principal components which is a significant information reduction since $q \ll p$ in most cases, typically $q = 2$ or $q = 3$. To measure the fraction of information retained in this q-dimensional space, we use the amount of variance $(\sum_{i=1}^{q} Var[Z_i])/(\sum_{i=1}^{p} Var[X_i])$ accounted for by these q principal components.

In this study the p original variables are the program characteristics mentioned in section 6.2. By examining the most important q principal components, which are linear combinations in the original program characteristics, meaningful interpretations can be given to these principal components in terms of the original program characteristics. To facilitate the interpretation of the principal components, we apply the *varimax* rotation [87]. This rotation makes the coefficients a_{ij} either close to ± 1 or zero, such that the original variables either have a strong impact on a principal component or they have no impact. Note that varimax rotation is an orthogonal transformation which implies that the rotated principal components are still uncorrelated.

The next step in the analysis is to display the various benchmarks as points in the q-dimensional space built up by the q principal components. This can be done by computing the values of the q retained principal components for each program-input pair. As such, a view can be given on the workload design space and the impact of input data sets on program behavior can be displayed, as will be discussed in the evaluation section of this chapter.

During principal components analysis, one can either work with normalized or non-normalized data (the data is normalized when the mean of each variable is zero and its variance is one). In the case of non-normalized data, a higher weight is given in the analysis to variables with a higher variance. In our experiments, we have used normalized data because of our heterogeneous data; e.g., the variance of the ILP is orders of magnitude larger than the variance of the data cache miss

rates.

6.3.2 Cluster analysis

Cluster analysis [87] is another data analysis technique that is aimed at clustering n cases, in our case program-input pairs, based on the measurements of p variables, in our case program characteristics. The final goal is to obtain a number of groups, containing program-input pairs that have 'similar' behavior. A commonly used algorithm for doing cluster analysis is *hierarchic clustering* which starts with a matrix of distances between the n cases or program-input pairs. As a starting point for the algorithm, each program-input pair is considered as a group. In each iteration of the algorithm, the two groups that are most close to each other (with the smallest distance, also called the *linkage distance*) will be combined to form a new group. As such, close groups are gradually merged until finally all cases will be in a single group. This can be represented in a so called *dendrogram*, which graphically represents the linkage distance for each group merge at each iteration of the algorithm. Having obtained a dendrogram, it is up to the user to decide how many clusters to take. This decision can be made based on the linkage distance. Indeed, small linkage distances imply strong clustering while large linkage distances imply weak clustering.

How we define the distance between two program-input pairs will be explained in the next section. To compute the distance between two groups, we have used the *nearest neighbour* strategy or *single linkage*. This means that the distance between two groups is defined as the smallest distance between two members of each group.

6.3.3 Workload analysis

The workload analysis done in this chapter is a combination of PCA and cluster analysis and consists of the following steps:

1. The $p = 20$ program characteristics as discussed in section 6.2 are measured by instrumenting the benchmark programs with ATOM [125], a binary instrumentation tool for the Alpha architecture. With ATOM, a statically linked binary can be transformed to an instrumented binary. Executing this instrumented binary with an appropriate input on an Alpha machine yields us the program

characteristics that will be used throughout the analysis. Measuring these $p = 20$ program characteristics was done for the 79 program-input pairs mentioned in section 6.4.1.

2. In a second step, these 79 (number of program-input pairs) $\times 20$ ($= p$, number of program characteristics) data points are normalized so that for each program characteristic the average equals zero and the variance equals one. On these normalized data points, principal components analysis (PCA) is done using STATISTICA [126], a package for statistical computations. This works as follows. A 2-dimensional matrix is presented as input to STATISTICA that has 20 columns representing the original program characteristics as mentioned in section 6.2. There are 79 rows in this matrix representing the various program-input pairs. On this matrix, PCA is performed by STATISTICA which yields us p principal components.

3. Once these p principal components are obtained, a varimax rotation can be done on these data for improving the understanding of the principal components. This can be done using STATISTICA.

4. Now, it is up to the user to determine how many principal components to be retained. This decision is made based on the amount of variance accounted for by the retained principal components.

5. The q principal components that are retained can be analyzed and a meaningful interpretation can be given to them. This is done based on the coefficients a_{ij}, also called the *factor loadings*, as they occur in the following equation $Z_i = \sum_{j=1}^{p} a_{ij} X_j$, with $Z_i, 1 \leq i \leq q$ the principal components and $X_j, 1 \leq j \leq p$ the original program characteristics. A positive coefficient a_{ij} means a positive impact of program characteristic X_j on principal component Z_i; a negative coefficient a_{ij} implies a negative impact. If a coefficient a_{ij} is close to zero, this means X_j has (nearly) no impact on Z_i.

6. The program-input pairs can be displayed in the workload space built up by these q principal components. This can be easily done by computing $Z_i = \sum_{j=1}^{p} a_{ij} X_j$ for each program-input pair.

7. Within this q-dimensional space the Euclidean distance can be computed between the various program-input pairs as a reliable

measure for the way program-input pairs differ from each other. There are two reasons supporting this statement. First, the values along the axes in this space are uncorrelated since they are determined by the principal components which are uncorrelated by construction. The absence of correlation is important when calculating the Euclidean distance because two correlated variables—that essentially measure the same thing—will contribute a similar amount to the overall distance as an independent variable; as such, these variables would be counted twice, which is undesirable. Second, the variance along the q principal components is meaningful since it is a measure for the diversity along each principal component by construction.

8. Finally, cluster analysis can be done using the distance between program-input pairs as determined in the previous step. Based on the dendrogram a clear view is given on the clustering within the workload space.

The reason why we chose to first perform PCA and subsequently cluster analysis instead of applying cluster analysis on the initial data is as follows. The original variables are highly correlated which implies that an Euclidean distance in this space is unreliable due to this correlation as explained previously. The most obvious solution would have been to use the *Mahalanobis* distance [87] which takes into account the correlation between the variables. However, the computation of the Mahalanobis distance is based on a pooled *estimate* of the common covariance matrix which might introduce inaccuracies.

6.4 Evaluation

In this section, we first present the program-input pairs that are used in this study. Second, we show the results of performing the workload analysis as discussed in section 6.3.3. Finally, the methodology is validated in section 6.4.3.

benchmark	input	dyn (M)	stat	mem (K)
gcc	amptjp	835	147,402	375
	c-decl-s	835	147,369	375
	cccp	886	145,727	371
	cp-decl	1,103	143,153	579
	dbxout	141	120,057	215
	emit-rtl	104	127,974	108
	explow	225	105,222	280
	expr	768	142,308	653
	gcc	141	129,852	125
	genoutput	74	117,818	104
	genrecog	100	124,362	133
	insn-emit	126	84,777	199
	insn-recog	409	105,434	357
	integrate	188	133,068	199
	jump	133	126,400	130
	print-tree	136	118,051	201
	protoize	298	137,636	159
	recog	227	123,958	161
	regclass	91	125,328	117
	reload1	778	146,076	542
	stmt-protoize	654	148,026	261
	stmt	356	138,910	250
	toplev	168	125,810	218
	varasm	166	139,847	168
postgres	Q2	227	57,297	345
	Q3	948	56,676	358
	Q4	564	53,183	285
	Q5	7,015	60,519	654
	Q6	1,470	46,271	1,080
	Q7	932	69,551	631
	Q8	842	61,425	11,821
	Q9	9,343	68,837	10,429
	Q10	1,794	62,564	681
	Q11	188	65,747	572
	Q12	1,770	65,377	258
	Q13	325	65,322	264
	Q14	1,440	67,966	448
	Q15	1,641	67,246	640
	Q16	82,228	58,067	389
	Q17	183	54,835	366

Table 6.1: Characteristics of the benchmarks (part one) used with their inputs, dynamic instruction count (in millions), static instruction count (number of instructions executed at least once) and data memory footprint in 64-bit words (in thousands).

benchmark	input	dyn (M)	stat	mem (K)
li	boyer	226	9,067	36
	browse	672	9,607	39
	ctak	583	8,106	18
	dderiv	777	9,200	16
	deriv	719	8,826	15
	destru2	2,541	9,182	16
	destrum2	2,555	9,182	16
	div2	2,514	8,546	19
	puzzle0	2	8,728	19
	tak2	6,892	8,079	16
	takr	1,125	8,070	36
	triang	3	9,008	15
ijpeg	band (2362x1570)	2,934	16,183	5,718
	beach (512x480)	254	16,039	405
	building (1181x1449)	1,626	16,224	2,742
	car (739x491)	373	16,294	596
	dessert (491x740)	353	16,267	587
	globe (512x512)	274	16,040	436
	kitty (512x482)	267	16,088	412
	monalisa (459x703)	259	16,160	508
	penguin (1024x739)	790	16,128	1,227
	specmun (1024x688)	730	15,952	1,136
	vigo (1024x768)	817	16,037	1,273
compress	14000000 e 2231 (ref)	60,102	4,507	4,601
	10000000 e 2231	42,936	4,507	3,318
	5000000 e 2231	21,495	4,494	1,715
	1000000 e 2231	4,342	4,490	433
	500000 e 2231	2,182	4,496	272
	100000 e 2231	423	4,361	142
m88ksim	train	24,959	11,306	4,834
	ref	71,161	14,287	4,834
vortex	train	3,244	78,766	1,266
	ref	92,555	78,650	5,117
perl	jumble	2,945	21,343	5,951
	primes	17,375	16,527	8
	scrabbl	28,251	21,674	4,098
go	50 9 2stone9.in	593	55,894	45
	50 21 9stone21.in	35,758	62,435	57
	50 21 5stone21.in	35,329	62,841	57

Table 6.2: Characteristics of the benchmarks (part two) used with their inputs, dynamic instruction count (in millions), static instruction count (number of instructions executed at least once) and data memory footprint in 64-bit words (in thousands).

6.4.1 Experimental setup

In this study, we have used the SPECint95 benchmarks[1] and a database workload consisting of TPC-D queries[2], see Tables 6.1 and 6.2. The reason why we chose SPECint95 instead of the more recent SPECint2000 is to limit the simulation time. SPEC opted to dramatically increase the runtimes of the SPEC2000 benchmarks compared to the SPEC95 benchmarks which is beneficial for performance evaluation on real hardware but impractical for simulation purposes. In addition, there are more reference inputs provided with SPECint95 then with SPECint2000. For gcc (GNU C compiler) and li (lisp interpreter), we have used all the reference input files. For ijpeg (image processing), penguin, specmun and vigo were taken from the reference input set. The other images that served as input to ijpeg were taken from the web. The dimensions of the images are shown in brackets. For compress (text compression), we have adapted the reference input '14000000 e 2231' to obtain different input sets. For m88ksim (microprocessor simulation) and vortex (object-oriented database), we have used the train and the reference inputs. The same was done for perl (perl interpreter): jumble was taken from the train input, and primes and scrabbl were taken from the reference input; as well as for go (game): '50 9 2stone9.in' from the train input, and '50 21 9stone21.in' and '50 21 5 stone21.in' from the reference input.

In addition to SPECint95, we used postgres v6.3 running the decision support TPC-D queries over a 100MB Btree-indexed database. For postgres, we ran all TPC-D queries except for query 1 because of memory constraints on our machine.

The benchmarks were compiled with optimization level -O4 and linked statically with the -non_shared flag for the Alpha architecture.

6.4.2 Results

In this section, we will first perform PCA on the data for the various input sets of gcc. Subsequently, the same will be done for postgres. Finally, PCA and cluster analysis will be applied on the data for all the benchmark-input pairs of Tables 6.1 and 6.2. We present the data for gcc and postgres before presenting the analysis of all the program-

[1] http://www.spec.org
[2] http://www.tpc.org

input pairs because these two benchmarks illustrate different aspects of the techniques in terms of the number of principal components, clustering, etc.

Gcc

PCA and varimax rotation extract two principal components from the data of gcc with 24 input sets. These two principal components together account for 96.9% of the total variance; the first and the second component account for 49.6% and 47.3% of the total variance, respectively. In Figure 6.1, the factor loadings are presented for these two principal components. E.g., this means that the first principal component is computed as $PC1 = 0.43 \times ILP + 0.94 \times bimodal + 0.94 \times gshare + \ldots$. The first component is positively dominated, see Figure 6.1, by the branch prediction accuracy, the percentage of arithmetic and logical operations; and negatively dominated by the I-cache miss rates. The second component is positively dominated by the D-cache miss rates, the percentage of shift and control operations; and negatively dominated by the ILP, the percentage of load/store operations and the number of instructions between two taken branches. Figure 6.2 presents the various input sets of gcc in the 2-dimensional space built up by these two components. Data points in this graph with a high value along the first component, have high branch prediction accuracies and high percentages of arithmetic and logical operations compared to the other data points; in addition, these data points also have low I-cache miss rates. Note that these data are normalized. Thus, only relative distances are important. For example, emit-rtl and insn-emit are relatively closer to each other than emit-rtl and cp-decl.

Figure 6.2 shows that gcc executing input set explow exhibits a different behavior than the other input sets. This is due to its high D-cache miss rates, its high percentage of shift and control operations, and its low ILP, its low percentage of load/store operations and its low number of instructions between two taken branches. The input sets emit-rtl and insn-emit have a high I-cache miss rate, a low branch prediction accuracy and a low percentage of arithmetic and logical operations; for reload1 the opposite is true. This can be concluded from the factor loadings presented in Figure 6.1; we also verified that this is true by inspecting the original data. The strong cluster in the middle of the graph contains the input sets gcc, genoutput, genrecog, jump, regclass, stmt and stmt-protoize. Note that although the characteristics mentioned in

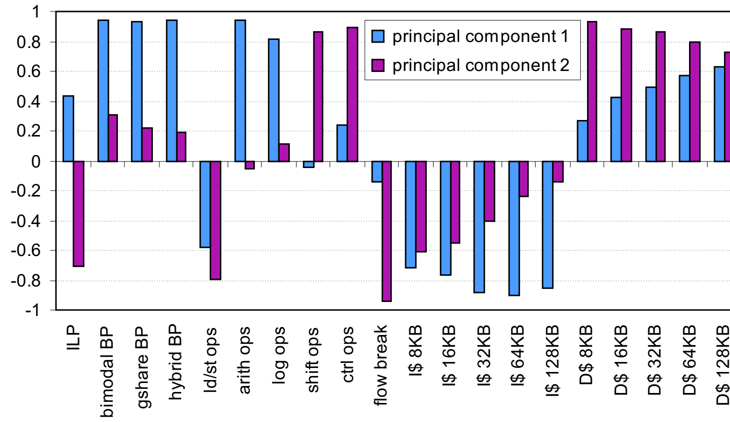


Figure 6.1: Factor loadings for gcc.

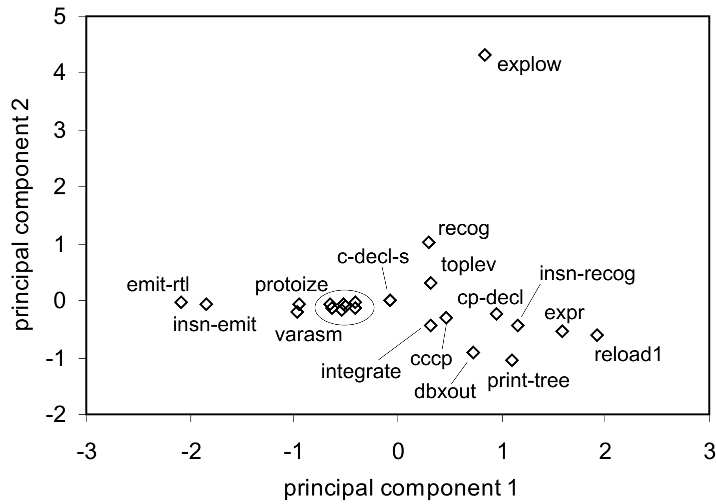


Figure 6.2: Workload space for gcc.

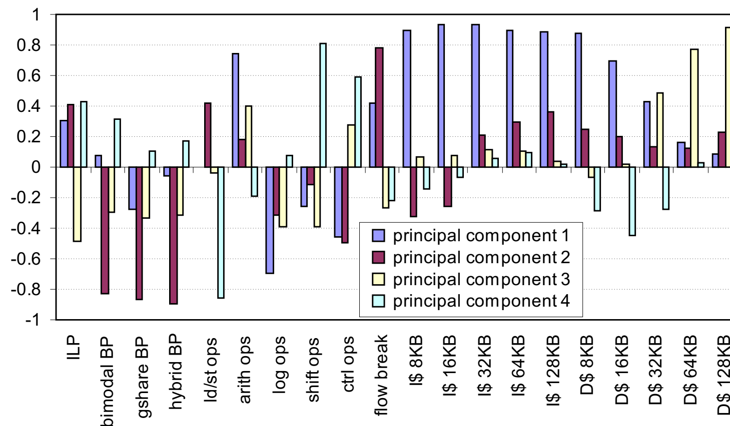


Figure 6.3: Factor loadings for postgres running the TPC-D queries.

Tables 6.1 and 6.2 (i.e., dynamic and static instruction count, and data memory footprint) are significantly different, these input sets result in a quite similar program behavior.

TPC-D

PCA extracted four principal components from the data of postgres running 16 TPC-D queries, accounting for 96.2% of the total variance; The first component accounts for 38.7% of the total variance and is positively dominated, see Figure 6.3, by the percentage of arithmetic operations, the I-cache miss rate and the D-cache miss rate for small cache sizes; and negatively dominated by the percentage of logical operations. The second component accounts for 24.7% of the total variance and is positively dominated by the number of instructions between two taken branches and negatively dominated by the branch prediction accuracy. The third component accounts for 16.3% of the total variance and is positively dominated by the D-cache miss rates for large cache sizes. The fourth component accounts for 16.4% of the total variance and is positively dominated by the percentage of shift operations and negatively dominated by the percentage memory operations.

Figure 6.4 shows the data points of postgres running the TPC-D queries in the 4-dimensional space built up by these four (rotated) components. To display this 4-dimensional space understandably, we have

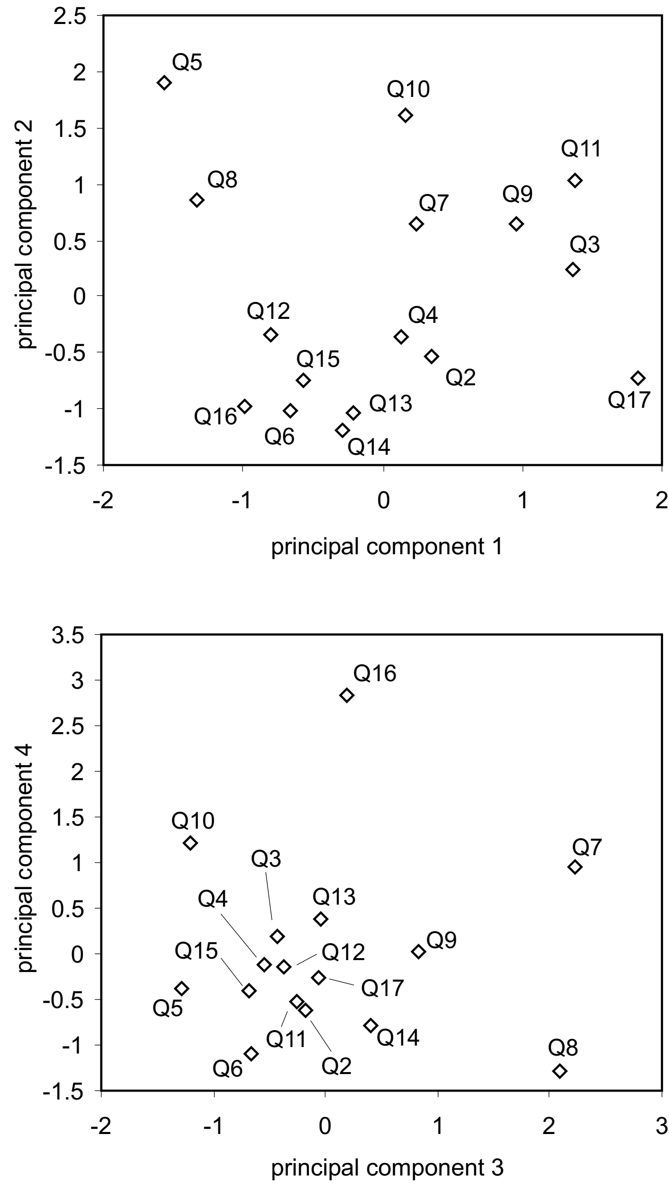


Figure 6.4: Workload space for postgres: first component vs. second component (top graph) and third vs. fourth component (bottom graph).

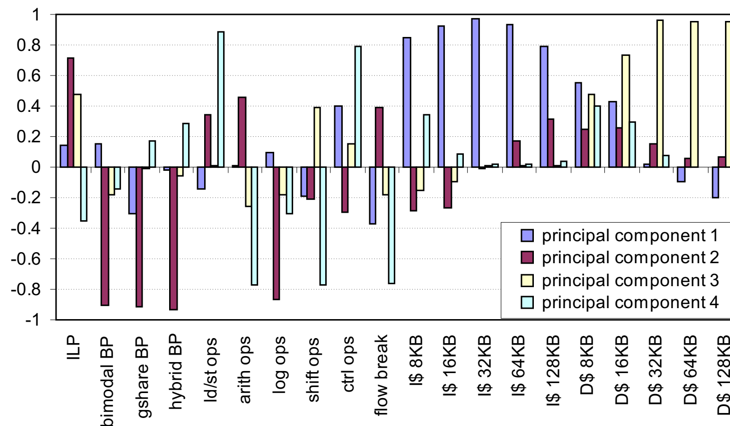


Figure 6.5: Factor loadings for all program-input pairs.

shown the first principal component versus the second in one graph; and the third versus the fourth in another graph. This graph does not reveal a strong clustering among the various queries. From this graph, we can also conclude that some queries exhibit a significantly different behavior than the other queries. For example, queries 7 and 8 have significantly higher D-cache miss rates for large cache sizes. Query 16 has a higher percentage of shift operations and a lower percentage of load/store operations.

Workload space

PCA extracts four principal components from the data of all 79 benchmark-input pairs as described in Tables 6.1 and 6.2, accounting for 93.1% of the total variance. The first component accounts for 26.0% of the total variance and is positively dominated, see Figure 6.5, by the I-cache miss rate. The second principal component accounts for 24.9% of the total variance and is positively dominated by the amount of ILP and negatively dominated by the branch prediction accuracy and the percentage of logical operations. The third component accounts for 21.3% of the total variance and is positively dominated by the D-cache miss rates. The fourth component accounts for 20.9% of the total variance and is positively dominated by the percentage of load/store and control operations and negatively dominated by the percentage of

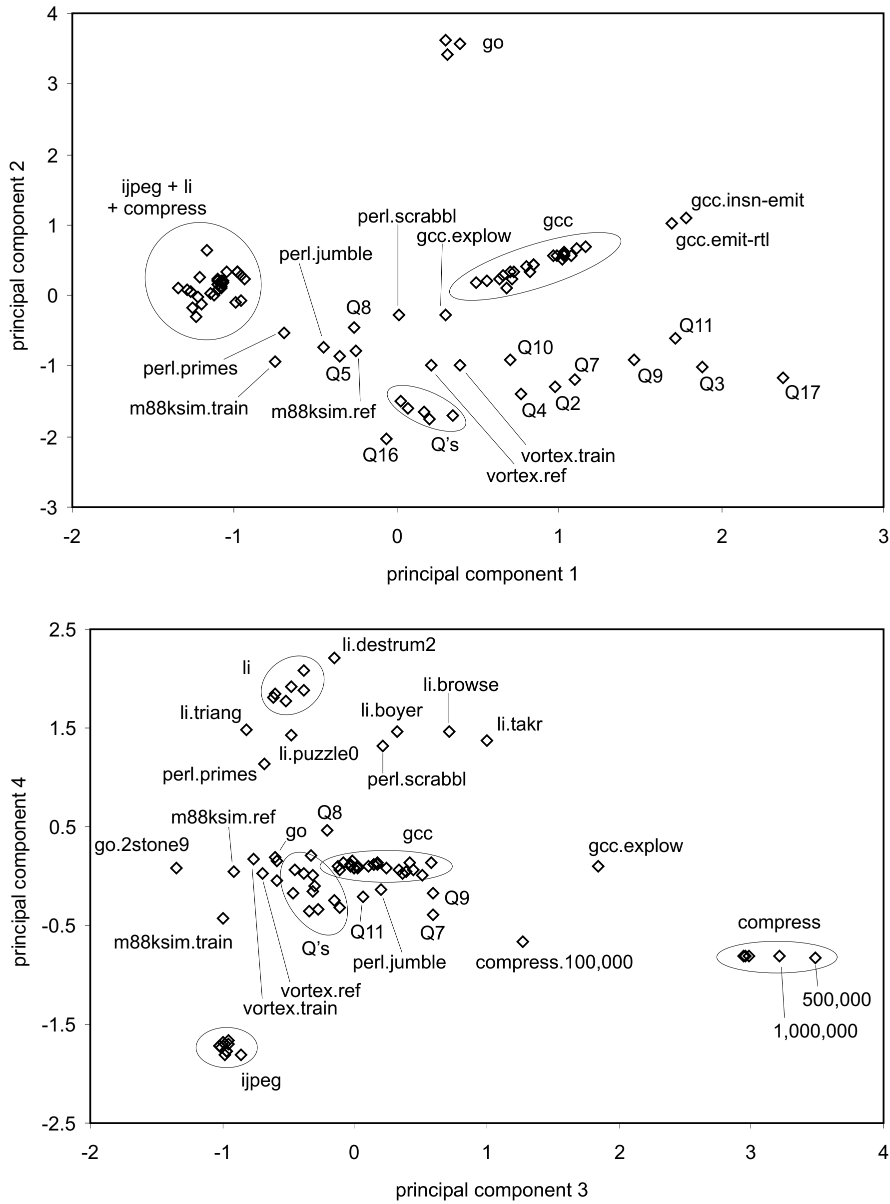


Figure 6.6: Workload space for all program-input pairs: first component vs. second component (upper graph) and third vs. fourth component (bottom graph).

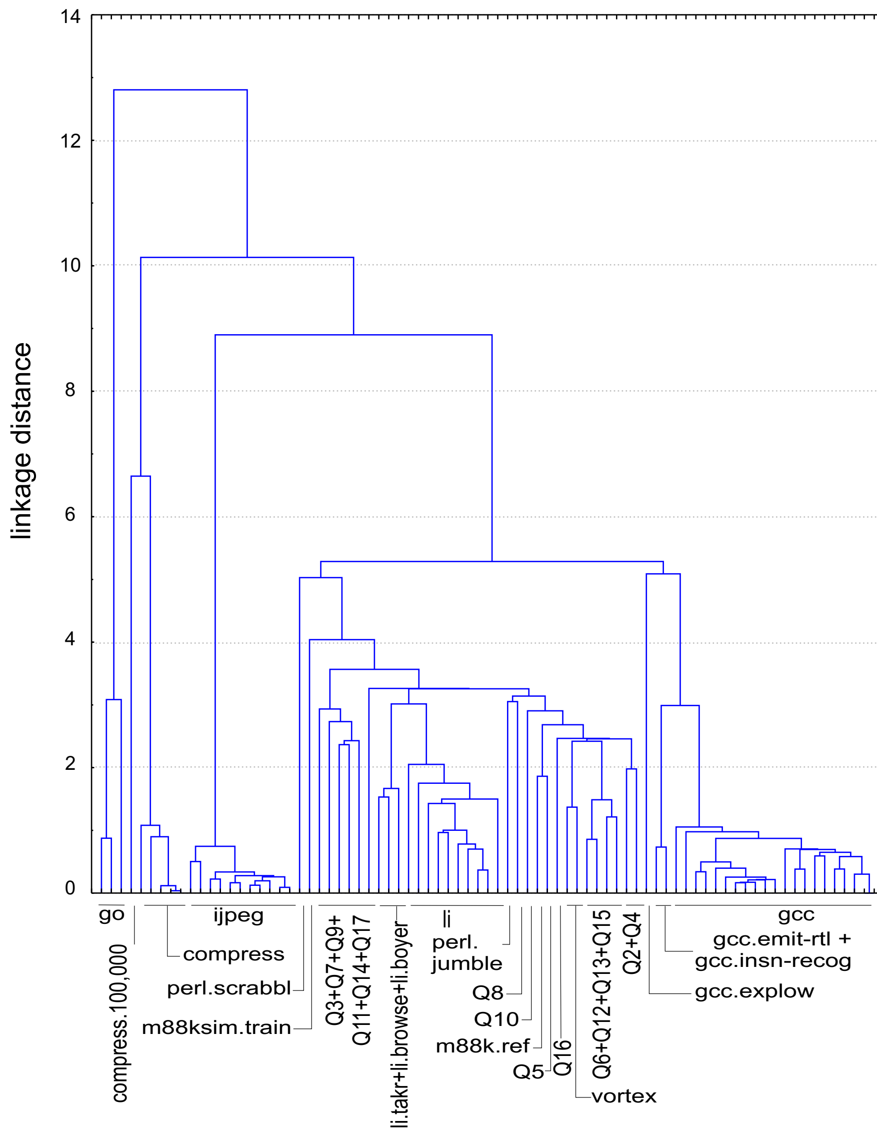


Figure 6.7: Cluster analysis.

arithmetic and shift operations as well as the number of instructions between two sequential flow breaks.

The results of the analyses that were done on these data, are shown in Figures 6.6 and 6.7. Figure 6.6 represents the program-input pairs in the 4-dimensional workload space built up by the four retained principal components. The dendrogram corresponding to the cluster analysis is shown in Figure 6.7. Program-input pairs connected by small linkage distances are clustered in early iterations of the analysis and thus, exhibit 'similar' behavior. Program-input pairs on the other hand, connected by large linkage distances exhibit different behavior.

Isolated points. From the data presented in Figures 6.6 and 6.7, it is clear that benchmarks go, ijpeg and compress are isolated in this 4-dimensional space. Indeed, in the dendrogram shown in Figure 6.7, these three benchmarks are connected to the other benchmarks through long linkage distances. E.g., go is connected to the other benchmarks with a linkage distance of 12.8 which is much larger than the linkage distance for more strongly clustered pairs, e.g., 2 or 4. An explanation for this phenomenon can be found in Figure 6.6. Compress discriminates itself along the third component which is due to its high D-cache miss rate. For ijpeg, the different behavior is due to, along the fourth component, the high percentage of arithmetic and shift operations, the high number of instructions between two taken branches and the low percentage of load/store and control operations. For go the discrimination is made along the second component or the low branch prediction accuracy, the low percentage of logical operations and the high amount of ILP.

Strong clusters. There are also several strong clusters which suggests that only a small number (or in some cases, only one) of the input sets should be selected to represent the whole cluster. This will ultimately reduce the total simulation time since only a few (or only one) program-input pairs need to be simulated instead of all the pairs within that cluster. We can identify several strong clusters:

- The data points corresponding to the gcc benchmark are strongly clustered, except for the input sets emit-rtl, insn-emit and explow. These three input sets exhibit a different behavior from the rest of the input sets. However, emit-rtl and insn-emit have a quite similar behavior.

- The data points corresponding to the lisp interpreter li except for browse, boyer and takr are strongly clustered as well. This can be clearly seen from Figure 6.7 where this group is clustered with a linkage distance that is smaller than 2. The three input sets with a different behavior are grouped with the other li input sets with a linkage distance of approximately 3. The variety within li is caused by the data cache miss rate measured by the third principal component, see Figure 6.6.

- According to Figure 6.7, there is also a small cluster containing TPC-D queries, namely queries 6, 12, 13 and 15.

- All input sets for ijpeg result in similar program behavior since all input sets are clustered in one group. An important conclusion from this analysis is that in spite of the differences in image dimensions, ranging from small images (512x482) to large images (2362x1570), the behavior of ijpeg remains quite the same.

- The input sets for compress are strongly clustered as well except for '100000 e 2231'.

Reference vs. train inputs. Along with its benchmark suite SPECint, SPEC releases reference and train inputs. The purpose for the train inputs is to provide input sets that should be used for profile-based compiler optimizations. The reference input is then used for reporting results. Within the context of this chapter, the availability of reference and train input sets is important for two reasons. First, when reference and train inputs result in similar program behavior we can expect that profile-driven optimizations will be effective. Second, train inputs have a smaller dynamic instruction counts which make them candidates for more efficient simulation runs. I.e., when a train input exhibits a similar behavior as a reference input, the train input can be used instead of the reference input for exploring the design space which will lead to a more efficient design flow.

In this respect, we take the following conclusions:

- The train and reference input for vortex exhibit similar program behavior.

- For m88ksim on the other hand, this is not true.

- For go, the train input '50 9 2stone9.in' leads to a behavior that is different from the behavior of the reference inputs '50 21 9stone21.in' and '50 21 5stone21.in'. The two reference inputs on the other hand, have a quite similar behavior.
- All three inputs for perl (two reference inputs and one train input) result in quite different behavior.

Reduced inputs. KleinOsowski *et al.* [79, 80] propose to reduce the simulation time of benchmarks by using reduced input sets. The final goal of their work is to identify a reduced input for each benchmark that results in similar behavior as the reference input but with a significant reduction in dynamic instruction counts and thus simulation time. From the data in Figures 6.6 and 6.7, we can conclude that, e.g., for ijpeg this is a viable option since small images result in quite similar behavior as large images. For compress on the other hand, we have to be careful: the reduced input '100000 e 2231' which was derived from the reference input '14000000 e 2231' results in quite different behavior. The other reduced inputs for compress lead to a behavior that is similar to the reference input.

Impact of input set on program behavior. As stated before, this analysis is useful for identifying the impact of input sets on program behavior. For example:

- The data points corresponding to postgres running the TPC-D queries are weakly clustered. The spread along the first principal component is very large and covers a large fraction of the first component. Therefore, a wide range of different I-cache behavior can be observed when running the TPC-D queries. Note also that all the queries result in an above-average branch prediction accuracy, a high percentage of logical operations and low ILP (negative value along the second principal component).
- The difference in behavior between the input sets for compress is mainly due to the difference in the data cache miss rates (along the third principal component).
- In general, the variation between programs is larger than the variation between input sets for the same program. Thus, when composing a workload, it is more important to select different pro-

grams with a well chosen input set than to include various inputs for the same program. For example, the program-input pairs for gcc (except for explow, emit-rtl and insn-emit) and ijpeg are strongly clustered in the workload space. In some cases however, for example postgres and perl, the input set has a relatively high impact on program behavior.

6.4.3 Preliminary validation

As stated before, the purpose of the analysis presented in this chapter is to identify clusters of program-input pairs that exhibit similar behavior. We will show that pairs that are close to each other in the workload space indeed exhibit similar behavior when changes are made to the microarchitecture on which they run.

In this section, we present a preliminary validation in which we observe the behavior of several input sets for gcc and one input set of each of the following benchmarks: go with 50 9 2stone9.in as input and li with boyer as input. The reason for doing a validation using a selected number of program-input pairs instead of all 79 program-input pairs is to limit simulation time. The simulations that are presented in this section already took several weeks. As a consequence, simulating all program-input pairs would have been impractically long[3]. However, since gcc presents a very diverse behavior (strong clustering versus isolated points, see Figure 6.2), we believe that a successful validation on gcc with some additional program-input pairs can be extrapolated to the complete workload space with confidence.

We have used seven input sets for gcc, namely explow, insn-recog, gcc, genoutput, stmt, insn-emit and emit-rtl. According to the analysis done in section 6.4.2, emit-rtl and insn-emit should exhibit a similar behavior; the same should be true for gcc, genoutput and stmt. explow and insn-recog on the other hand, should result in a different program behavior since they are quite far away from the other input sets that are selected for this analysis.

We used SimpleScalar v3.0 [18] for the Alpha architecture as simulation tool for this analysis. The baseline architecture has a window size of 64 instructions and an issue width of 4.

In Figures 6.8, 6.9, 6.10 and 6.11, the number of instructions retired per cycle (IPC) is shown as a function of the I-cache configuration, the

[3] This is exactly the problem we are trying to solve.

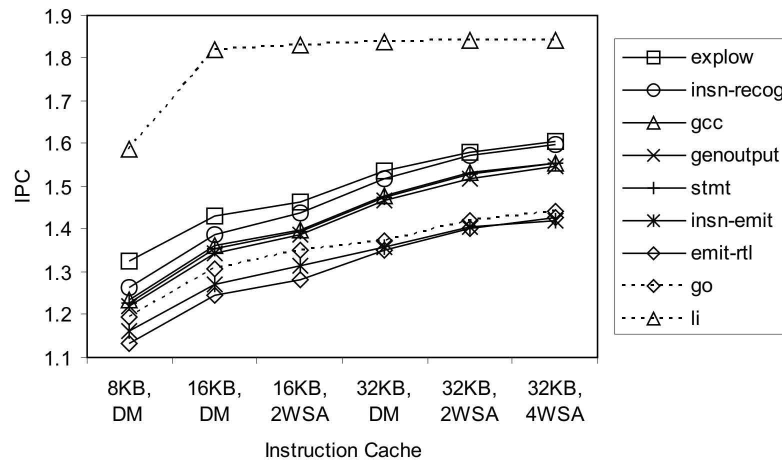


Figure 6.8: IPC as a function of the I-cache configuration; 16KB DM D-cache and 8K-entry bimodal branch predictor.

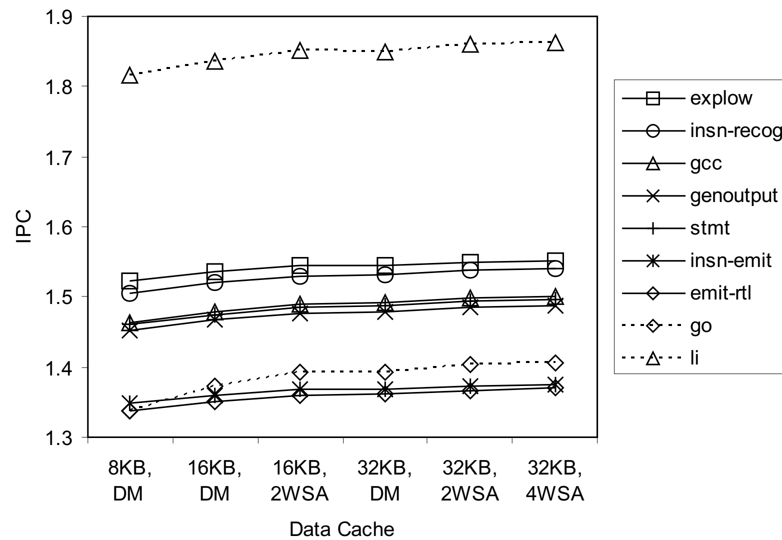


Figure 6.9: IPC as a function of the D-cache configuration; 32KB DM I-cache and 8K-entry bimodal branch predictor.

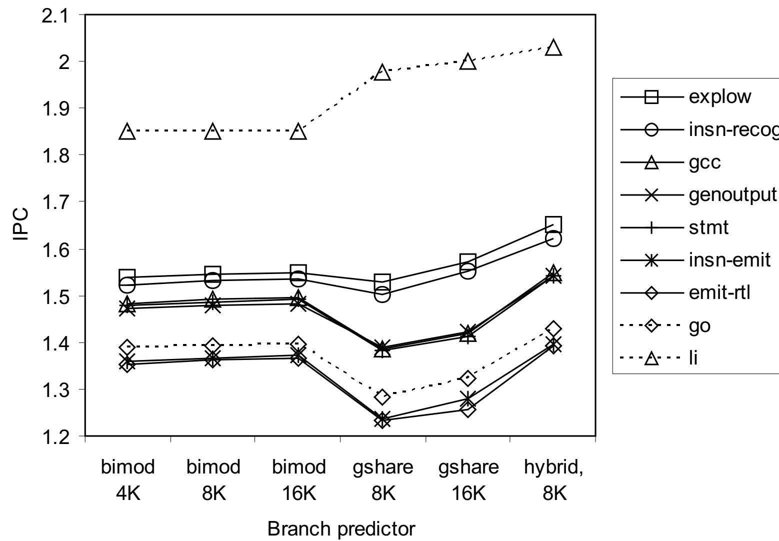


Figure 6.10: IPC a function of the branch predictor configuration; 32KB DM I-cache and 32KB DM D-cache.

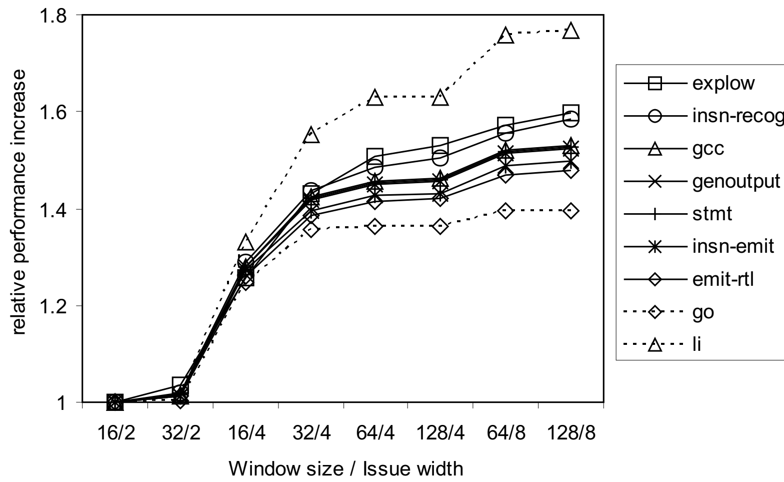


Figure 6.11: Performance increase w.r.t. the 16/2 configuration as a function of the window size and the issue width; 32KB DM I-cache, 32KB DM D-cache and 8K-entry branch predictor.

D-cache configuration, the branch predictor and the window size versus issue width configuration, respectively. We will first discuss the results for gcc. Afterwards, we will detail on the other benchmarks.

For gcc, we clearly identify three groups of input sets that have similar behavior, namely (i) explow and insn-recog, (ii) gcc, genoutput and stmt, and (iii) insn-emit and emit-rtl. For example, in Figure 6.10, the branch behavior of group (i) is significantly different from the other input sets. Or, in Figure 6.11, the scaling behavior as a function of window size and issue width is quite different for all three groups. This could be expected for groups (ii) and (iii) as discussed earlier. The fact that explow and insn-recog exhibit similar behavior on the other hand, is unexpected since these two input sets are quite far away from each other in the workload space, see Figure 6.2. The discrimination between these two input sets is primarily along the second component. Along the first component on the other hand, explow and insn-recog have a similar value. This leads us to the conclusion that the impact on performance of the program characteristics measured along the second principal component is smaller than along the first component.

The other two benchmarks, go and li, clearly exhibit a different behavior on all four graphs. This could be expected from the analysis done in section 6.4.2 since PCA and cluster analysis pointed out that these benchmarks have a different behavior. Most of the mutual differences can be explained from the analysis done in section 6.4.2. For example, li has a different I-cache behavior than gcc and go which is reflected in Figure 6.8. Also, go has a different D-cache behavior than gcc which is clearly reflected in Figure 6.9. Other differences however, are more difficult to explain. Again, this phenomenon is due to the fact that some microarchitectural parameters have a minor impact on performance for a given microarchitectural configuration. However, for other microarchitectural configurations we can still expect different behavior. For example, go has a different branch behavior than gcc, according to the analysis done in section 6.4.2; in Figure 6.10, go and gcc exhibit the same behavior.

6.5 Related work

Saavedra and Smith [114] addressed the problem of measuring benchmark similarity. For this purpose they presented a metric that is based on dynamic program characteristics for the Fortran language, for ex-

ample the instruction mix, the number of function calls, the number of address computations, etc. For measuring the difference between benchmarks they used the squared Euclidean distance. The methodology in this chapter differs from the one presented by Saavedra and Smith [114] for two reasons. First, the program characteristics measured here are more suited for performance prediction of contemporary architectures since we include branch prediction accuracy, cache miss rates, ILP, etc. Second, we prefer to work with uncorrelated program characteristics (obtained after PCA) for quantifying differences between program-input pairs, as extensively argued in section 6.3.3.

Hsu *et al.* [68] studied the impact of input data sets—in this casus test, train and reference inputs for SPECint2000—on program behavior using high-level metrics, such as procedure level profiles and IPC, as well as low-level metrics, such as the execution paths leading to data cache misses. They conclude that the test input sets are not suitable to be used for simulation. The train input set on the other hand, was found to be better; however, for some benchmarks, train input sets might lead to misleading performance results.

KleinOsowski *et al.*[79, 80] propose to reduce the simulation time of the SPEC 2000 benchmark suite by using reduced input data sets. Instead of using the reference input data sets provided by SPEC, which result in unreasonably long simulation times, they propose to use smaller input data sets that accurately reflect the behavior of the full reference input sets. For determining whether two input sets result in more or less the same behavior, they used the chi-squared statistic based on the function-level execution profiles for each input set. Note that a resemblance of function-level execution profiles does not necessarily imply a resemblance of other program characteristics which are probably more directly related to performance, such as instruction mix, cache behavior, etc. The latter approach was taken in this chapter for exactly that reason. KleinOsowski *et al.* also recognized that this is a potential problem. The methodology presented in this chapter can be used as well for selecting reduced input data sets. A reference input set and a resembling reduced input set will be situated close to each other in the q-dimensional space built up by the principal components.

Another important research topic that is related to this chapter is trace sampling [24, 30, 76, 84]. In trace sampling, several samples are taken from a program execution so that the total number of instructions in the samples is significantly less than the total number of instruc-

tions of a complete execution. In order to make viable design decisions based on these sampled traces, a sampled trace should be representative for the complete program execution. Iyengar *et al.* [72] propose an R-metric for measuring the representativeness of a sampled trace. Lafage and Seznec [82] propose to choose representative samples using a data reduction technique, namely cluster analysis. The methodology presented here can also be used to validate sampled traces. Indeed, a sampled trace that is situated close to its reference trace in the workload space could be considered as being representative. Trace sampling and reduced input sets are compared in [63].

Statistical simulation is related to the research topic presented in this chapter, since the success of both techniques relies on choosing relevant program characteristics to be incorporated in the analysis. For statistical simulation, relevant program characteristics are needed to obtain a high accuracy; for the technique presented in this chapter, relevant program characteristics are needed to construct a reliable workload space.

Another possible application of using a data reduction technique such as principal components analysis, is to compare different workloads. In [21], Chow *et al.* used PCA to compare the branch behavior of Java and non-Java workloads. The interesting aspect of using PCA in this context is that PCA is able to identify on which point two workloads differ.

Huang and Shen [69] evaluated the impact of input data sets on the bandwidth requirements of computer programs.

Changes in program behavior due to different input data sets are also important for profile-guided compilation [120], where profiling information from a past run is used by the compiler to guide its optimizations. Fisher and Freudenberger [53] studied whether branch directions from previous runs of a program (using different input sets) are good predictors of the branch directions in future runs. Their study concludes that branches generally take the same directions in different runs of a program. However, they warn that some runs of a program exercise entirely different parts of the program. Hence, these runs cannot be used to make predictions about each other. By using the average branch direction over a number of runs, this problem can be avoided. Wall [135] studied several types of profiles such as basic block counts and the number of references to global variables. He measured the usefulness of a profile as the speedup obtained when that profile is used in

a profile-guided compiler optimization. Seemingly, the best results are obtained when the same input is used for profiling and measuring the speedup. This implies that every input is different in some sense and leads to different compiler optimizations.

6.6 Conclusion

In microprocessor design, it is important to have a representative workload to make correct design decisions. This chapter proposes the use of principal components analysis and cluster analysis to efficiently explore the workload space. In this workload space, benchmark-input pairs can be displayed and a distance can be computed that gives us an idea of the behavioral difference between these benchmark-input pairs. This representation can be used to measure the impact of input data sets on program behavior. In addition, our methodology was successfully validated by showing that program-input pairs that are close to each other in the principal components space, indeed exhibit similar behavior as a function of microarchitectural changes. Interesting applications for this technique are the composition of workloads and profile-based compiler optimizations.

Chapter 7

Conclusion

Time discovers truth.
Seneca

In this chapter, we summarize the conclusions that can be taken from this dissertation. In addition, we highlight interesting research topics that need to be further investigated in the future.

7.1 Summary

As stated in the introduction of this dissertation, there are a number of issues involved with the earliest stages of the microprocessor design process, see section 1.1. In the following enumeration we discuss how this dissertation contributes to each of these issues. First, we should have representative workloads. In other words, the program-input pairs included in the workload should be carefully chosen so that the workload is representative for the target environment of operation of the microprocessor. In chapter 6 we have proposed a technique to measure the impact of input data sets on the behavior of computer programs that is based on multivariate data analysis techniques, such as principal components analysis and cluster analysis. One of the applications of this method is the composition of representative workloads.

Second, the design process should be sped up to reduce the time to market without compromising the final design, i.e., the early design

stage methods should be reasonably accurate so that viable design decisions are made. A major part of this dissertation evaluates the statistical simulation methodology in the context of performance estimation, see chapters 2, 3 and 4. Statistical simulation was found to be a fast simulation technique—only requiring the simulation of a limited number of synthetic instructions, e.g., 1 million instructions, instead of several hundreds or thousands of millions—while yielding quite accurate performance predictions. The absolute prediction errors were found to be smaller than 15% to 20% in general for the individual benchmarks; the average prediction error is 10% for the SPECint95 traces and 6% for the IBS traces. The relative accuracy was found to be very good which is even more important than the absolute accuracy when doing design space explorations. We discussed what the implications are of guaranteeing syntactical correctness of the synthetic traces. We showed that in order to preserve syntactical correctness and at the same time guarantee a correct implementation of the desired dependency distance distribution in the synthetic trace, a feedback loop should be implemented in the synthetic trace generator. We also demonstrated that the accuracy can be improved significantly by modeling clustered I-cache misses. Another important contribution of this dissertation is the observation that register traffic characteristics exhibit a power law distribution. This notion can be used in an analytical workload model that allows to explore the entire workload design space whereas a collection of benchmarks only contains isolated points from the workload space. In conclusion, we can state that statistical simulation can be used to identify a region of interest in the (huge) design space that can be further explored using more detailed, thus more accurate and thus slower architectural simulations using real program traces. Consequently, the total design time will be significantly reduced.

Third, the impact of chip technology, such as interconnect delays and power consumption, should be included in the earliest stages of the design since these aspects of technology are extremely important in current and near future chip technologies. In this dissertation we have shown that statistical simulation can deal with both of these. For example, statistical simulation is capable of accurately quantifying the impact on performance of inserting additional pipeline stages in the frontend of the pipeline (which increases the branch prediction penalty) or in the backend of the pipeline (which increases the L1 D-cache access time). As such, statistical simulation could be used in combination with an early floorplanner to investigate the impact of microar-

chitectural design decisions on cycle time and vice versa. Concerning statistical simulation in combination with power modeling, we have shown in chapter 5 that power modeling through statistical simulation is highly accurate and can thus be used for searching regions of energy efficient microarchitectures in the entire design space. In addition, the combination of statistical simulation with power modeling allows the investigation of the interaction of program characteristics and energy consumption per cycle.

7.2 Future work

The future belongs to those who prepare for it.
American proverb

The important thing is not to stop questioning.
Curiosity has its own reason for existing.
Albert Einstein

We can present several topics for future research.

- A first interesting area of research could be to further improve the performance prediction accuracy of statistical simulation. Indeed, although statistical simulation was found to be quite accurate in this dissertation, there is still room for improvement. We can make several suggestions:

  - by using more detailed statistical profiles that include more correlation between the various program characteristics. As discussed in section 3.3, Nussbaum and Smith [102] show that higher-order distributions increase the absolute accuracy of statistical simulation. Next to increasing the accuracy, this research is interesting for another purpose as well, namely for getting insight in the behavior of computer programs, more specifically for getting insight in the interactions between the various program characteristics.

- by measuring different statistical profiles for different program phases. It is shown in [116] that computer programs exhibit several phases with different I-cache behavior, D-cache behavior, branch prediction behavior, ILP, etc. As such, using separate statistical profiles for each program phase can potentially improve the accuracy of statistical simulation. The statistical simulation method presented in this dissertation on the other hand, uses a statistical profile that is averaged over the complete program run.
- by using different statistical profiles for instructions along misspeculated paths. As said in section 2.5, we model resource contention due to branch mispredictions by injecting synthetically generated instructions in the pipeline that are labeled as coming from the misspeculated path. These instructions and their characteristics are generated synthetically using a statistical profile that is measured over the instructions along the program execution path (thus assuming perfect branch prediction). However, these characteristics can be quite different from the characteristics of the instructions along the misspeculated paths. As such, this is a potential source of error. This intuition can be further supported by noting that not accurately modeling instructions along misspeculated paths in a trace-driven simulator can lead to significant prediction errors of up to 12% according to the analysis done by Combs, Combs and Shen [22].
- by modeling load latency tolerance. As suggested in section 3.2, an important program characteristic that is not modeled in the current statistical simulation methodology is whether load operations are latency tolerant. Load latency tolerance [124] means that for some loads delaying its execution does not affect the overall performance; for other loads on the other hand, performance is highly affected by the execution latency of the load. Explicitly modeling load latency and correlating this information with the dependency graph and D-cache miss behavior is thus a potential improvement that needs further investigation.

• Another possible research direction is to search for additional application domains for the statistical simulation method. Possible application domains are:

  - statistical simulation in combination with early floorplanning. As discussed previously, early floorplanning is a useful tool in an early design stage to estimate the impact of microarchitectural design decisions on the chip layout, the interconnect structure and by consequence the interconnect delay between the various structures on the chip. Combining these estimates with statistical simulation could be very useful for making design decisions in the earliest stages of the design that take into account IPC (by using the results as generated in chapter 2), timing issues (through early floorplanning) and power modeling (see chapter 5).
  - system design through statistical simulation. As discussed in chapter 2, the usefulness of statistical simulation is even bigger in the area of system design where one single system consists of a number of individual components that need to be simulated simultaneously. Consequently, the simulation problem is even more prominent in this area than in the area of uniprocessor design. A first evaluation of statistical simulation for system design was done by Nussbaum and Smith [103] for the evaluation of symmetric multiprocessor systems.

- Another interesting research topic is to compare statistical simulation with other early design stage techniques or fast simulation techniques, for example trace sampling, as discussed in section 3.3. In this analysis several aspects need to be taken into account, namely absolute and relative accuracy, simulation speed, complexity, flexibility, etc.

- Concerning workload design, see chapter 6, it would be interesting to further investigate the following issues:
  - evaluating the impact of the program characteristics included in the analysis on the results of the analysis.
  - evaluating the representativeness of sampled traces or reduced input sets for the SPEC2000 suite [79, 80] using this methodology[1].

[1] This future work was completed in the period between submitting this dissertation to the PhD committee and the PhD defense. A paper describing an evaluation of the reduced input sets for the SPEC2000 suite is accepted for publication in IEEE Computer [45]. A paper extending [48] in which the representativeness of sampled traces is

– evaluating the usefulness of this methodology in the context of profile-guided compilation.

evaluated, is tentatively accepted for the Journal of Instruction-Level Parallelism [46].

Bibliography

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[2] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA-19)*, pages 342–351, May 1992.

[3] C. Bechem, J. Combs, N. Utamaphetai, B. Black, R. D. Shawn Blanton, and J. P. Shen. An integrated functional performance simulator. *IEEE Micro*, 19(3):26–35, May/June 1999.

[4] R. Bhargava, L. K. John, and F. Matus. Accurately modeling speculative instruction fetching in trace-driven simulation. In *Proceedings of the IEEE Performance, Computers, and Communications Conference (IPCCC-1999)*, pages 65–71, February 1999.

[5] B. Black, A. S. Huang, M. H. Lipasti, and J. P. Shen. Can trace-driven simulators accurately predict superscalar performance? In *Proceedings of the 1996 International Conference on Computer Design (ICCD-96)*, October 1996.

[6] B. Black and J. P. Shen. Calibration of microprocessor performance models. *IEEE Computer*, 31(5):59–65, May 1998.

[7] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, July/August 1999.

[8] P. Bose. Performance test case generation for microprocessors. In *Proceedings of the 16th IEEE VLSI Test Symposium*, pages 50–58, April 1998.

[9] P. Bose. Performance evaluation and validation of microprocessors. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'99)*, pages 226–227, May 1999.

[10] P. Bose. Performance evaluation of processor operation using trace pre-processing. US Patent 6,059,835, May 2000.

[11] P. Bose and T. M. Conte. Performance analysis and its impact on design. *IEEE Computer*, 31(5):41–49, May 1998.

[12] P. Bose, T. M. Conte, and T. M. Austin. Challenges in processor modeling and validation. *IEEE Micro*, 19(3):9–14, May 1999.

[13] D. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, November/December 2000.

[14] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA-7)*, pages 171–182, January 2001.

[15] D. Brooks, M. Martonosi, and P. Bose. Abstraction via separable components: An empirical study of absolute and relative accuracy in processor performance modeling. Technical Report RC 21909, IBM Research Division, T. J. Watson Research Center, December 2000.

[16] D. Brooks, M. Martonosi, J.-D. Wellman, and P. Bose. Power-performance modeling and tradeoff analysis for a high end microprocessor. In *Proceedings of the Power-Aware Computer Systems (PACS'00) held in conjunction with ASPLOS-IX*, November 2000.

[17] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, pages 83–94, June 2000.

[18] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set. Computer Architecture News, 1997. See also `http://www.simplescalar.com` for more information.

[19] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power at high performance. In *Proceedings of the Power-Aware Computer Systems (PACS'00) held in conjunction with ASPLOS-IX*, November 2000.

[20] R. Carl and J. E. Smith. Modeling superscalar processors via statistical simulation. In *Workshop on Performance Analysis and its Impact on Design (PAID-98), held in conjunction with the 25th Annual International Symposium on Computer Architecture (ISCA-25)*, June 1998.

[21] K. Chow, A. Wright, and K. Lai. Characterization of Java workloads by principal components analysis and indirect branches. In *Proceedings of the Workshop on Workload Characterization (WWC-1998), held in conjunction with the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-31)*, pages 11–19, November 1998.

[22] J. Combs, C. B. Combs, and J. P. Shen. Mispredicted path cache effects. In *Proceedings of the 1999 Euro-Par Conference*, pages 1322–1331, August 1999.

[23] T. M. Conte. *Systematic Computer Architecture Prototyping*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[24] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design (ICCD-96)*, October 1996.

[25] P. Crowley and J.-L. Baer. Trace sampling for desktop applications on windows NT. In *Proceedings of the First Workshop on Workload Characterization (WWC-1998) held in conjunction with the 31st ACM/IEEE Annual International Symposium on Microarchitecture (MICRO-31)*, November 1998.

[26] H. J. Curnow and B. A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, 1976.

[27] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA-28)*, pages 266–277, July 2001.

[28] A. Dhodapkar, C. H. Lim, G. Cai, and W. R. Daasch. TEM^{2}P^{2}EST: A thermal enabled multi-model power/performance estimator. In *Proceedings of the Power-Aware Computer Systems (PACS'00) held in conjunction with ASPLOS-IX*, November 2000.

[29] P. K. Dubey, G. B. Adams III, and M. J. Flynn. Instruction window size trade-offs and characterization of program parallelism. *IEEE Transactions on Computers*, 43(4):431–442, April 1994.

[30] P. K. Dubey and R. Nair. Profile-driven sampled trace generation. Technical Report RC 20041, IBM Research Division, T. J. Watson Research Center, April 1995.

[31] L. Eeckhout, T. Vander Aa, B. Goeman, H. Vandierendonck, R. Lauwereins, and K. De Bosschere. Application domains for fixed-length block structured architectures. In *Proceedings of the Sixth Australasian Computer Systems Architecture Conference (ACSAC 2001)*, pages 35–44, January 2001.

[32] L. Eeckhout and K. De Bosschere. Statistical simulation of superscalar architectures using commercial workloads. In *Fourth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-4) held in conjunction with the Seventh International Symposium on High Performance Computer Architecture (HPCA-7)*, January 2001.

[33] L. Eeckhout and K. De Bosschere. Quantifying behavioral differences between multimedia and general-purpose workloads. *Journal of Systems Architecture*, 2003. Accepted for publication.

[34] L. Eeckhout, K. De Bosschere, and H. Neefs. On the feasibility of fixed-length block structured architectures. In *Proceedings of the 5th Australasian Computer Architecture Conference ACAC 2000*, pages 17–25, January 2000.

[35] L. Eeckhout and K. De Bosschere. Nonuniform behavior in instruction traces for contemporary processors. In *AIP Conference Proceedings of the Fourth International Conference on Computing Anticipatory Systems (CASYS'2000)*, pages 662–672, August 2000.

[36] L. Eeckhout and K. De Bosschere. Early design phase power/performance modeling through statistical simulation. In

Proceedings of the 2001 International IEEE Symposium on Performance Analysis of Systems and Software (ISPASS-2001), pages 10–17, November 2001.

[37] L. Eeckhout and K. De Bosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*, pages 25–34, September 2001.

[38] L. Eeckhout and K. De Bosschere. Increasing the accuracy of statistical simulation for modeling superscalar processors. In *The 20th IEEE International Performance, Computing and Communications Conference (IPCCC 2001)*, pages 196–204, April 2001.

[39] L. Eeckhout, K. De Bosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *The IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2000)*, pages 1–6, April 2000.

[40] L. Eeckhout, H. Neefs, and K. De Bosschere. Estimating IPC of a block structured instruction set architecture in an early design stage. In *Parallel Computing: Fundamentals and Applications; Proceedings of the International Conference ParCo99*, pages 468–475, January 2000.

[41] L. Eeckhout, H. Neefs, K. De Bosschere, and J. Van Campenhout. Improving loop performance on a block structured architecture through predication. In *Proceedings of the Tenth International Conference on Parallel and Distributed Computing and Systems*, pages 457–462, October 1998.

[42] L. Eeckhout, H. Neefs, K. De Bosschere, and J. Van Campenhout. Investigating the implementation of a block structured processor architecture in an early design stage. In *Proceedings of the 25th Euromicro Conference*, volume 1, pages 186–193, September 1999.

[43] L. Eeckhout, H. Neefs, K. De Bosschere, and J. Van Campenhout. On the organisation and implementation of a fixed-length block structured instruction set architecture. In *Proceedings of the 1999 ACM/IEEE International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 77–82, March 1999.

[44] L. Eeckhout, H. Neefs, and K. De Bosschere. Early design stage exploration of fixed-length block structured architectures. *Journal of Systems Architecture*, 46:1469–1486, December 2000.

[45] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Designing workloads for computer architecture research. *IEEE Computer*, 2003. Accepted for publication.

[46] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 2003. `http://www.jilp.org`. Tentatively accepted for publication.

[47] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. How input data sets change program behaviour. In *Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-02) held in conjunction with the Eighth International Symposium on High-Performance Computer Architecture (HPCA-8)*, February 2002.

[48] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Workload design: Selecting representative program-input pairs. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT-2002)*, pages 83–94, September 2002.

[49] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *IEEE Computer*, 35(2):68–76, February 2002.

[50] P. G. Emma. Understanding some simple processor-performance limits. *IBM Journal of Research and Development*, 41(3):215–232, May 1997.

[51] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*, pages 149–159, December 1997.

[52] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA-21)*, pages 211–222, April 1994.

[53] J.A. Fisher and S.M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 85–95, 1992.

[54] M. J. Flynn, P. Hung, and K. W. Rudd. Deep-submicron microprocessor design issues. *IEEE Micro*, 19(4):11–22, July/August 1999.

[55] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 22nd Annual International Symposium on Microarchitecture (MICRO-22)*, pages 236–245, December 1992.

[56] M. Franklin and G. S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[57] S. Ghiasi and D. Grunwald. A comparison of two architectural power models. In *Proceedings of the Power-Aware Computer Systems (PACS'00) held in conjunction with ASPLOS-IX*, November 2000.

[58] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich. FLASH vs. (simulated) FLASH: Closing the simulation loop. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 49–58, November 2000.

[59] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. In *Proceedings of the 35th Design Automation Conference (DAC-1998)*, pages 726–731, June 1998.

[60] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14):1–6, October 1996.

[61] A. Hartstein and T. R. Puzak. The optimal pipeline depth for a microprocessor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA-29)*, pages 7–13, May 2002.

[62] J. W. Haskins Jr. and K. Skadron. Minimal subset evaluation: Rapid warm-up for simulated hardware state. In *Proceedings of*

the 2001 International Conference on Computer Design (ICCD-2001), pages 32–39, September 2001.

[63] J. W. Haskins Jr., K. Skadron, A. J. KleinOsowski, and D. J. Lilja. Techniques for accurate, accelerated processor simulation: An analysis of reduced inputs and sampling. Technical Report CS-2002-01, University of Virginia—Dept. of Computer Science, January 2002.

[64] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, third edition, 2003.

[65] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[66] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q1 2001.

[67] C. Hsieh and M. Pedram. Micro-processor power estimation using profile-driven program synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(11):1080–1089, November 1998.

[68] W. C. Hsu, H. Chen, P. Y. Yew, and D.-Y. Chen. On the predictability of program behavior using different input data sets. In *Proceedings of the Sixth Workshop on Interaction between Compilers and Computer Architectures (INTERACT 2002), held in conjunction with the Eighth International Symposium on High-Performance Computer Architecture (HPCA-8)*, pages 45–53, February 2002.

[69] A. S. Huang and J. P. Shen. The intrinsic bandwidth requirements of ordinary programs. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 105–114, October 1996.

[70] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: Simulating shared-memory multiprocessors with ILP processors. *IEEE Computer*, 35(2):40–49, February 2002.

[71] V. S. Iyengar and L. H. Trevillyan. Evaluation and generation of reduced traces for benchmarks. Technical Report RC 20610, IBM Research Division, T. J. Watson Research Center, October 1996.

[72] V. S. Iyengar, L. H. Trevillyan, and P. Bose. Representative traces for processor models with infinite cache. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture (HPCA-2)*, pages 62–73, February 1996.

[73] L. K. John, P. Vasudevan, and J. Sabarinathan. Workload characterization: Motivation, goals and methodology. In L. K. John and A. M. G. Maynard, editors, *Workload Characterization: Methodology and Case Studies*. IEEE Computer Society, 1999.

[74] N. P. Jouppi. The nonuniform distribution of instruction-level and machine parallelism and its effect on performance. *IEEE Transactions on Computers*, 38(12):1645–1658, December 1989.

[75] R. A. Kamin III, G. B. Adams III, and P. K. Dubey. Dynamic trace analysis for analytic modeling of superscalar performance. *Performance Evaluation*, 19(2-3):259–276, March 1994.

[76] R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 43(6):664–675, June 1994.

[77] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 microprocessor architecture. In *Proceedings of the 1998 International Conference on Computer Design (ICCD-98)*, pages 90–95, October 1998.

[78] H. Khalid. Validating trace-driven microarchitectural simulations. *IEEE Micro*, 20(6):76–82, November/December 2000.

[79] A. J. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja. Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research. In *Workload Characterization of Emerging Computer Applications, Proceedings of the IEEE 3rd Annual Workshop on Workload Characterization (WWC-2000) held in conjunction with the International Conference on Computer Design (ICCD-2000)*, pages 83–100, September 2000.

[80] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. Computer Architecture Letters, June 2002.

[81] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the Eighth Annual International Symposium on Computer Architecture (ISCA-8)*, pages 81–87, May 1981.

[82] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *IEEE 3rd Annual Workshop on Workload Characterization (WWC-2000) held in conjunction with the International Conference on Computer Design (ICCD-2000)*, September 2000.

[83] S. Laha, J. H. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11):1325–1336, November 1988.

[84] G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. Technical Report SMLI TR-93-22, Sun Microsystems Laboratories Inc., December 1993.

[85] G. Loh. A time-stamping algorithm for efficient performance estimation of superscalar processors. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS-2001)*, pages 72–81, May 2001.

[86] P. S. Magnusson, M. Christensson, Jesper Eskilson, D. Forsgren, G. Hallberg, J. Högberg nad F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.

[87] B. F. J. Manly. *Multivariate Statistical Methods: A primer*. Chapman & Hall, second edition, 1994.

[88] M. Martonosi, A. Gupta, and T. Anderson. Effectiveness of trace sampling for performance debugging tools. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 248–259, May 1993.

[89] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. In *Proceedings of the 2002 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 108–116, June 2002.

[90] S. McFarling. Combining branch predictors. Technical Report WRL TN-36, Digital Western Research Laboratory, June 1993.

[91] S. Melvin and Y. Patt. Enhancing instruction scheduling with a block-structured ISA. *International Journal of Parallel Programming*, 23(3):221–243, 1995.

[92] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT-1999)*, pages 2–10, October 1999.

[93] M. Moudgill. Techniques for implementing fast processor simulators. In *Proceedings of the 31st Annual Simulation Symposium*, pages 83–90, April 1998.

[94] M. Moudgill, J.-D. Wellman, and J. H. Moreno. Environment for PowerPC microarchitecture exploration. *IEEE Micro*, 19(3):15–25, May/June 1999.

[95] V. Narayananan, D. LaPotin, R. Gupta, and G. Vijayan. PEPPER – a timing driven early floorplanner. In *Proceedings of the 1995 International Conference on Computer Design: VLSI in Computers and Processors (ICCD-1995)*, pages 230–235, October 1995.

[96] H. Neefs. A preliminary study of a fixed length block structured instruction set architecture. Technical Report Paris 96-07, Dept. of Electronics and Information Systems, Ghent University, November 1996.

[97] H. Neefs, K. De Bosschere, and J. Van Campenhout. Exploitable levels of ILP in future processors. *Journal of Systems Architecture*, 45(9):687–708, March 1999.

[98] H. Neefs, H. Vandierendonck, and K. De Bosschere. A technique for high bandwidth and deterministic low latency load/store accesses to multiple cache banks. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture (HPCA-6)*, pages 313–324, January 2000.

[99] A.-T. Nguyen, P. Bose, K. Ekanadham, A. Nanda, and M. Michael. Accuracy and speed-up of parallel trace-driven architectural simulation. In *Proceedings of the 11th International Parallel Processing Symposium (IPPS'97)*, pages 39–44, April 1997.

[100] D. B. Noonburg and J. P. Shen. Theoretical modeling of superscalar processor performance. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*, pages 52–62, November 1994.

[101] D. B. Noonburg and J. P. Shen. A framework for statistical modeling of superscalar processor performance. In *Proceedings of the third International Symposium on High-Performance Computer Architecture (HPCA-3)*, pages 298–309, February 1997.

[102] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*, pages 15–24, September 2001.

[103] S. Nussbaum and J. E. Smith. Statistical simulation of symmetric multiprocessor systems. In *Proceedings of the 35th Annual Simulation Symposium 2002*, pages 89–97, April 2002.

[104] D. J. Ofelt. *Efficient Performance Prediction for Modern Microprocessors*. PhD thesis, Stanford University, August 1999.

[105] D. J. Ofelt and J. L. Hennessy. Efficient performance prediction for modern microprocessors. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS-2000)*, pages 229–239, June 2000.

[106] M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, pages 71–82, June 2000.

[107] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-24)*, pages 206–218, June 1997.

[108] A. Poursepanj. The PowerPC performance modeling methodology. *Communications of the ACM*, 37(6):47–55, June 1994.

[109] N. Ranganathan and M. Franklin. An empirical study of decentralized ILP execution models. In *Proceedings of the 8th International Conference on Architectural Support for Programming Lan-*

guages and Operating Systems (ASPLOS-VIII), pages 272–281, October 1998.

[110] M. Reilly. Designing an Alpha microprocessor. *IEEE Computer*, 32(7):27–34, July 1999.

[111] M. Reilly and J. Edmondson. Performance simulation of an Alpha microprocessor. *IEEE Computer*, 31(5):50–58, May 1998.

[112] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(1):78–103, January 1997.

[113] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, December 1997.

[114] R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, November 1996.

[115] E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memoization. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 283–294, October 1998.

[116] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*, pages 3–14, September 2001.

[117] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques. *IEEE Transactions on Computers*, 48(11):1260–1281, November 1999.

[118] M. Smelyanski, E. S. Davidson, and D. Burger. Fast and accurate performance modeling of out-of-order issue processors. Unpublished; available at `http://www.eecs.umich.edu/~msmelyan`, March 2001.

[119] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, December 1995.

[120] M. D. Smith. Overcoming the challenges to feedback-directed optimization (keynote talk). In *Proceedings of ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 1–11, 2000.

[121] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA-22)*, pages 414–425, June 1995.

[122] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic evaluation of shared-memory systems with ILP processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-25)*, pages 380–391, June 1998.

[123] M. S. Squillante, D. R. Kaeli, and H. Sinha. Analytic models of workload behavior and pipeline performance. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTTS-1997)*, pages 91–96, January 1997.

[124] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. *Journal of Instruction-Level Parallelism*, 1, October 1999. http://www.jilp.org/vol1.

[125] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Technical Report 94/2, Western Research Lab, Compaq, March 1994.

[126] StatSoft, Inc. STATISTICA for Windows. Computer program manual. 1999. `http://www.statsoft.com`.

[127] H. S. Stone. *High-Performance Computer Architecture*. Addison-Wesley Publishing Company, 1987.

[128] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the 1993 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'93)*, pages 24–35, 1993.

[129] D. Thiébaut. From the fractal dimension of the intermiss gaps to the cache-miss ratio. *IBM Journal of Research and Development*, 32(6):796–803, November 1988.

[130] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal*, 11:25–33, January 1967.

[131] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Instruction fetching: Coping with code bloat. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA-22)*, pages 345–356, June 1995.

[132] T. Vander Aa, L. Eeckhout, B. Goeman, H. Vandierendonck, T. Van Achteren, R. Lauwereins, and K. De Bosschere. Optimizing a 3D image reconstruction algorithm: Investigating the interaction between the high-level implementation, the compiler and the architecture. In *Proceedings of the Seventh Asia-Pacific Computer Systems Architecture Conference (ACSAC-2002)*, pages 119–126, February 2002.

[133] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, pages 95–106, June 2000.

[134] J. Voldman, B. Mandelbrot, L. W. Hoevel, J. Knight, and P. Rosenfeld. Fractal nature of software-cache interaction. *IBM Journal of Research and Development*, 27(2):164–170, March 1983.

[135] D. W. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of the 1991 International Conference on Programming Language Design and Implementation (PLDI-1991)*, pages 59–70, 1991.

[136] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, October 1984.

[137] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *Proceedings of the 1991 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 79–89, May 1991.

[138] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of SimplePower: A cycle-accurate energy estimation tool. In *Proceedings of the 37th Design Automation Conference*, pages 340–345, June 2000.

[139] K. C. Yeager. MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2), April 1996.

[140] V. V. Zyuban and P. M. Kogge. Inherently low-power high-performance superscalar architectures. *IEEE Transactions on Computers*, 50(3):268–285, March 2001.