BENCHMARKING MODERN MULTIPROCESSORS

CHRISTIAN BIENIA

A DISSERTATION

PRESENTED TO THE FACULTY

OF PRINCETON UNIVERSITY

IN CANDIDACY FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE

BY THE DEPARTMENT OF

COMPUTER SCIENCE

ADVISOR: KAI LI

JANUARY 2011

Abstract

Benchmarking has become one of the most important methods for quantitative performance evaluation of processor and computer system designs. Benchmarking of modern multiprocessors such as chip multiprocessors is challenging because of their application domain, scalability and parallelism requirements. In my thesis, I have developed a methodology to design effective benchmark suites and demonstrated its effectiveness by developing and deploying a benchmark suite for evaluating multiprocessors.

More specifically, this thesis includes several contributions. First, the thesis shows that a new benchmark suite for multiprocessors is needed because the behavior of modern parallel programs is significantly different from those represented by SPLASH-2, the most popular parallel benchmark suite developed over ten years ago. Second, the thesis quantitatively describes the requirements and characteristics of a set of multithreaded programs and their underlying technology trends. Third, the thesis presents a systematic approach to scale and select benchmark inputs with the goal of optimizing benchmarking accuracy subject to constrained execution or simulation time. Finally, the thesis describes a parallel benchmark suite called PARSEC for evaluating modern shared-memory multiprocessors. Since its initial release, PARSEC has been adopted by many architecture groups in both research and industry.

Acknowledgments

First and foremost I would like to acknowledge the great support of my advisor, Kai Li. His academic advice and patience with me was instrumental to allow my research to make the progress which it has made. Most of all I am grateful for his ability to connect me with the right people at the right time, which is a key factor for the success of a project that is as interdisciplinary as the development of a new benchmark suite.

I would like to acknowledge the many authors of the PARSEC benchmark programs which are too numerous to be listed here. The institutions who contributed the most number of programs are Intel and Princeton University. Stanford University allowed me to use their code and data for the `facesim` benchmark. Many researchers submitted patches which were included in the PARSEC distribution.

During my time at Princeton University I also received tremendous support from many faculty members and graduate students. They gave my advice on possible benchmark programs from their own research area and allowed me to use their software projects where we identified opportunities for new workloads. Not all of their work could be included in the final version of the benchmark suite.

I would like to explicitly acknowledge the contributions of the following individuals: Justin Rattner, Pradeep Dubey, Tim Mattson, Jim Hurley, Bob Liang, Horst Haussecker, Yemin Zhang, Ron Fedkiw and the scientists of the Application Research Lab of Intel. They convinced skeptics and supported me so that a project the size of PARSEC could succeed.

This work was supported in parts by the Gigascale Systems Research Center, the Intel Research Council, National Science Foundation grants CSR-0509447, CSR-0509402 and SGER-0849512, and by Princeton University.

To my parents, Heinrich and Wanda Bienia, who have supported me the entire time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

At the heart of the scientific method lies the experiment. This modern understanding of science was first developed and presented by Sir Isaac Newton in his groundbreaking work *Philosophiae naturalis principia mathematica* in 1687. Newton suggested that researchers are to develop hypotheses which they are then to test with observable, empirical and measurable evidence. It was the first time that a world view was formulated that was free of metaphysical considerations. Until today science follows Newton's methodology.

Experience shows that the strength of a scientific proof is no better than the strength of the scientific experiment used to produce it. A flawed methodology will usually result in flawed conclusions. This makes it crucial for researchers to develop and use experimental setups that are sound and rigorous enough to serve as a foundation for their work.

In computer science research the standard method to conduct scientific experiments is benchmarking. Scientists decide on a selection of benchmark programs which they then study in detail to arrive at generalized conclusions that typically apply to actual computer systems. Unless the chosen workloads allow us to generalize to a wider range of software the results obtained this way will only be of very limited validity. It is therefore necessary that the selection of benchmarks is a sufficiently accurate representation of the actual programs of interest.

My thesis makes the following contributions:

- Demonstrates that an overhauled selection of benchmark programs is needed. The behavior of modern workloads is significantly different from those represented by SPLASH-2, the most popular parallel benchmark suite.

- Quantitatively describes the characteristics of modern programs which allows one to sketch out the properties of future multiprocessor systems.
- Proposes a systematic approach to scale and select benchmark inputs. The presented methodology shows how to create benchmark inputs with varying degrees of accuracy and speed and how to select inputs from a given selection to maximize benchmarking accuracy subject to an execution time budget.
- Creates the Princeton Application Repository for Shared-Memory Computers (PARSEC), a new benchmark suite that represents modern shared-memory multithreaded programs using the methodologies developed during the course of my research. The suite has become a popular choice for researchers around the world.

My thesis is structured as follows: A basic overview of the work and its impact is given in Chapter 1. It describes which requirements a modern benchmark suite should satisfy, and why existing selections of benchmarks fall short of them. Chapter 2 presents the PARSEC benchmark suite, a new program selection which satisfies these criteria. Its description was previously published in [10, 12]. Chapter 3 compares PARSEC to SPLASH-2 and demonstrates that a new benchmark suite is indeed needed [9, 13]. A detailed characterization of PARSEC is presented in Chapter 4. Its contents was published as the PARSEC summary paper [11]. Chapter 5 discusses aspects relevant for the creation and selection of benchmark inputs [14]. The thesis concludes and describes future work in Chapter 6.

1.1 Purpose of Benchmarking

Benchmarking is a method to analyze computer systems. This is done by studying the execution of the benchmark programs, which are a representation of the programs of interest. Benchmarking requires that the behavior of the selected workloads is a sufficiently accurate description of the programs of interest, which means that the resulting instruction streams must be representative.

Unfortunately it is extremely challenging to prove that a benchmark is representative because the amount of information that is known about the full range of programs of interest is limited. In cases where a successful argument for representativeness can be

made the range of applications of interest is often very narrow to begin with and can even be as small as the benchmark program itself. For general-purpose benchmarking two approaches are typically taken:

Representative Program Selection A benchmark suite aims to represent a wide selection of programs with a small subset of benchmarks. By its very nature a benchmark suite is therefore a statistical sample of the application space. Creating a selection of benchmarks by choosing samples of applications in this top-down fashion can yield accurate representations of the program space of interest if the sample size is sufficiently high. However, it usually is impossible to make any form of hard statements about the representativeness of the suite because the application space covered is typically too large and not fully observable. For this reason a qualitative argument is usually made to establish credibility for these types of program selections. It is common that an interesting and important program cannot be used directly as a benchmark due to legal issues. In these cases the program is usually substituted with a proxy workload that implements the same algorithms. Proxy workloads are an important method to increase the coverage of a benchmark suite, but compared to real programs they offer less certainty because their characteristics might not be identical to the ones of the original program.

Diverse Range of Characteristics The futility to assess the representativeness of a benchmark suite has motivated approaches that focus on recreating the possible program behavior in the form of different combinations of characteristics. The advantage of this bottom-up method is that program characteristics are quantities that can be measured and compared. The full characteristics space is therefore significantly easier to describe and to analyze systematically. An example for this method of creating benchmarks are synthetic workloads that emulate program behavior to artificially create the desired characteristics. The major limitation of this method is that not all portions of the characteristics space are equally important, which means that design decisions and other forms of trade-offs can easily become biased towards program behavior that has minor importance in practice.

The work presented in this thesis follows both approaches to establish the presented methodology. The qualitative argument that the PARSEC benchmark suite covers emerging application domains better than existing benchmarks is made in Chapter 2. Chapter 3

shows that the program selection is at least as diverse as existing best-practice benchmarking methodologies by using quantitative methods to compare the characteristics of PARSEC with SPLASH-2.

1.2 Motivation

One goal of this work is to define a benchmarking methodology that can be used to drive the design of the new generation of multiprocessors and to make it available in the form of a benchmark suite that can be readily used by other researchers. Its workloads should serve as high-level problem descriptions which push existing processor designs to their limit. This section first presents the requirements for such a suite. It then discusses how existing benchmarks fail to meet these requirements.

1.2.1 Requirements for a Benchmark Suite

A multithreaded benchmark suite should satisfy the following five requirements:

Multithreaded Applications Shared-memory chip multiprocessors are already ubiquitous. The trend for future processors is to deliver large performance improvements through increasing core counts while only providing modest serial performance improvements. Consequently, applications that require additional processing power will need to be parallel.

Emerging Workloads Rapidly increasing processing power is enabling a new class of applications whose computational requirements were beyond the capabilities of the earlier generation of processors [24]. Such applications are significantly different from earlier applications (see Section 1.3). Future processors will be designed to meet the demands of these emerging applications.

Diversity Applications are increasingly diverse, run on a variety of platforms and accommodate different usage models. They include interactive applications such as computer games, offline applications such as data mining programs and programs with different parallelization models. Specialized collections of benchmarks can be used to study some of these areas in more detail, but decisions about general-purpose processors should be based on a diverse set of applications.

State-of-Art Algorithms A number of application areas have changed dramatically over the last decade and use very different algorithms and techniques. Visual applications for example have started to increasingly integrate physics simulations to generate more realistic animations [38]. A benchmark should not only represent emerging applications but also use state-of-art algorithms and data structures.

Research Support A benchmark suite intended for research has additional requirements compared to one used for benchmarking real machines alone. Benchmark suites intended for research usually go beyond pure scoring systems and provide infrastructure to instrument, manipulate, and perform detailed simulations of the included programs in an efficient manner.

1.2.2 Limitations of Existing Benchmark Suites

Existing benchmark suites fall short of the presented requirements and must thus be considered inadequate for evaluating modern CMP performance.

SPLASH-2 SPLASH-2 is a suite composed of multithreaded applications [96] and hence seems to be an ideal candidate to measure performance of CMPs. However, its program collection is skewed towards HPC and graphics programs. It does not include parallelization models such as the pipeline model which are used in other application areas. SPLASH-2 should furthermore not be considered state-of-art anymore. `Barnes` for example implements the Barnes-Hut algorithm for N-body simulation [7]. For galaxy simulations it has largely been superseded by the TreeSPH [36] method, which can also account for mass such as dark matter which is not concentrated in bodies. However, even for pure N-body simulation `barnes` must be considered outdated. In 1995 Xu proposed a hybrid algorithm which combines the hierarchical tree algorithm and the Fourier-based Particle-Mesh (PM) method to the superior TreePM method [98]. My analysis shows that similar issues exist for a number of other applications of the suite including `raytrace` and `radiosity`.

SPEC CPU2006 and OMP2001 SPEC CPU2006 and SPEC OMP2001 are two of the largest and most significant collections of benchmarks. They provide a snapshot of current scientific and engineering applications. Computer architecture research, however, commonly focuses on the near future and should thus also con-

sider emerging applications. Workloads such as systems programs and parallelization models which employ the producer-consumer model are not included. SPEC CPU2006 is furthermore a suite of serial programs that is not intended for studies of parallel machines.

Other Benchmark Suites Besides these major benchmark suites, several smaller workload collections exist. They were designed to study a specific program area and are thus limited to a single application domain. Therefore they usually include a smaller set of applications than a diverse benchmark suite typically offers. Due to these limitations they are commonly not used for scientific studies which do not restrict themselves to the covered application domain. Examples for these types of benchmark suites are ALPBench [55], BioParallel [41], MediaBench [53], MineBench [65], the NAS Parallel Benchmarks [3] and PhysicsBench [99]. Because of their different focus I do not discuss these suites in more detail.

1.3 Emerging Applications

Emerging applications play an important role for computer architecture research because they define the frontier of software development, which usually offers significant research opportunities. Changes on the software side often lead to comparable changes on the hardware side. An example for this duality is the emergence of graphics processing units (GPUs) that followed after the transition to 3D graphics by computer games.

The use of emerging applications as benchmark programs allows hardware designers to work on the same type of high-level problems as software developers. Emerging applications have often been studied less intensely than established programs due to their high degree of novelty. Changes in computational trends, user requirements or potential problems are more likely to be identified while studying emerging workloads.

An emerging application is simply a new type of software. There are several reasons why an emerging application might not have been widely available earlier:

- The application implements new algorithms or concepts which became available only recently due to a breakthrough in research.

- Changes in user demands or behavior made seemingly irrelevant workloads significantly more popular. This might be a consequence of disruptive changes in other areas.
- The emerging application has high computational demands that made its use infeasible. This is often the case with workloads that have a real-time requirement such as video games or other interactive programs.

1.4 History and Impact of PARSEC

The PARSEC research project started sometime in the fall of 2005. The trend to chip multiprocessor designs was already in full swing. PARSEC was created to allow the study of this increasingly popular architecture form with modern, emerging programs that were not taken from the HPC domain. The lack of contemporary consumer workloads was a significant issue at that time.

The initiative gained momentum during a project meeting sometime in 2006. It was Affiliates Day in the Computer Science Department of Princeton University. Among the visitors to the department was Justin Rattner, CTO of Intel, who joined us for the meeting. In the discussion that subsequently developed it became clear that Intel saw exactly the same problems in current benchmarking methodology. Moreover, Intel had a similar project to develop next-generation benchmark programs. Their suite was called *RMS benchmark suite* due to its focus on recognition, mining and synthesis of data. It was quickly agreed on to merge the research efforts and develop a single suite suitable for modern computer architecture research.

An early alpha version of the PARSEC suite was released internally on May 31, 2007. It contained mostly programs developed by Princeton University and the open source community. The acronym PARSEC was decided on in the days before the alpha release. It was the first time an early version of the benchmark suite took shape under this name. A beta version which already had most of the form of the final version followed a few months later on October 22, 2007.

On January 25, 2008, the first public release of PARSEC became generally available and was instantly adopted by the researchers who started putting it to use for their work. It had 12 workloads which made it through the selection process. The second version

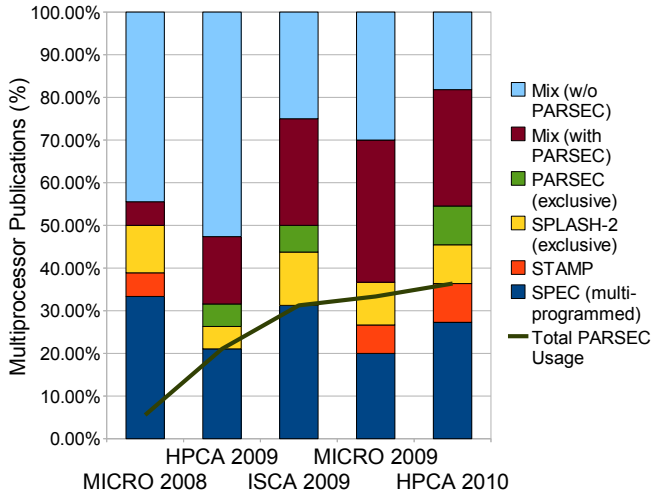

Figure 1.1: Usage of PARSEC at top-tier computer architecture conferences. The figure shows which benchmark suites were used for evaluations of shared-memory multiprocessor machines. At the beginning of 2010 36% of all publications in this area were already using PARSEC.

was published about a year later, on February 13, 2009. It improved several workloads significantly, implemented more alternative threading models and added one new workload. A maintenance version with several important bugfixes, PARSEC 2.1, was released six months later, on August 13, 2009.

Since its first initial release the PARSEC benchmark suite could establish itself as a welcome addition to the workloads used by other researchers. By now the PARSEC distribution was downloaded over 4,000 times. Figure 1.1 gives a breakdown of the types of benchmarks used for shared-memory multiprocessor evaluations at top-tier computer architecture conferences during the last two years. As can be seen the total usage of PARSEC has consistently risen and has already reached 36% of all analyzed publications. The PARSEC summary paper [11] was cited in more than 280 publications.

1.5 Conclusions

This thesis describes a holistic approach to benchmarking that covers the entire spectrum of work necessary to create and select workloads for performance experiments, starting from the initial program selection over the creation of accurate benchmark inputs to the final selection of a subset of workloads for the experiment. Previous work on benchmarks

was often limited to only a description of the programs and a basic analysis of their characteristics.

This thesis improves over previous work in the following ways:

- Demonstrates that a new benchmark suite is needed. Modern workloads come from emerging areas of computing and have significantly different characteristics compared to more established programs. Benchmark creators and users need to consider current trends in computing when they select their workloads. The choice of a set of benchmarks can influence the conclusions drawn from their measurement results.

- Characterizes a selection of modern multithreaded workloads. The analysis results allow to give quantitative estimates for the nature and requirements of contemporary multiprocessor workloads as a whole. This information makes it possible to describe the properties of future multiprocessors.

- Presents quantitative methods to create and select inputs for the benchmarks. Input data is a necessary component of every workload. Its choice can have significant impact on the observed characteristics of the program, but until now no systematic way to create and analyze inputs has been known. The methods presented in this thesis consider this fact and describe how to create and choose inputs in a way that minimizes behavior anomalies and maximizes the accuracy of the benchmarking results.

- Describes the PARSEC benchmark suite. PARSEC was created to make the research results developed during the course of this work readily available to other scientists. The overall goal of the suite is to give researchers freedom of choice for their experiments by providing them with a range of benchmarks, program inputs and workload features and the tools to leverage them for their work in an easy and intuitive way.

This combination of results allows the definition of a benchmarking methodology that gives scientists a higher degree of confidence in their experimental results than with previous approaches. The presented concepts and methods can be reused to create new and entirely different benchmark suites that are unrelated to PARSEC.

Chapter 2

The PARSEC Benchmark Suite

2.1 Introduction

Benchmarking is the quantitative foundation of computer architecture research. Benchmarks are used to experimentally determine the benefits of new designs. However, to be relevant, a benchmark suite needs to satisfy a number of properties. First, the applications in the suite should consider a target class of machines such as the multiprocessors that are the focus of this work. This is necessary to ensure that the architectural features being proposed are relevant and not obviated by minor rewrites of the application. Second, the benchmark suite should represent important applications on the target machines. The suite presented in this chapter will focus on emerging applications. Third, the workloads in the benchmark suite should be diverse enough to exhibit the range of behavior of the target applications. Finally, it is important that the programs use state-of-art algorithms and that the suite supports the work of its users, which is research for the suite presented here.

As time passes, the relevance of a benchmark suite diminishes. This happens not only because machines evolve and change over time but also because new applications, algorithms, and techniques emerge. New benchmark suites become necessary after significant changes in the architectures or applications.

In fact, dramatic changes have occurred both in mainstream processor designs as well as applications in the last few years. The arrival of chip multiprocessors (CMPs) with ever increasing number of cores has made parallel machines ubiquitous. At the same

Program	Application Domain	Parallelization		Working Set	Data Usage	
		Model	Granularity		Sharing	Exchange
blackscholes	Financial Analysis	data-parallel	coarse	small	low	low
bodytrack	Computer Vision	data-parallel	medium	medium	high	medium
canneal	Engineering	unstructured	fine	unbounded	high	high
dedup	Enterprise Storage	pipeline	medium	unbounded	high	high
facesim	Animation	data-parallel	coarse	large	low	medium
ferret	Similarity Search	pipeline	medium	unbounded	high	high
fluidanimate	Animation	data-parallel	fine	large	low	medium
freqmine	Data Mining	data-parallel	medium	unbounded	high	medium
raytrace	Rendering	data-parallel	medium	unbounded	high	low
streamcluster	Data Mining	data-parallel	medium	medium	low	medium
swaptions	Financial Analysis	data-parallel	coarse	medium	low	low
vips	Media Processing	data-parallel	coarse	medium	low	medium
x264	Media Processing	pipeline	coarse	medium	high	high

Table 2.1: Qualitative summary of the inherent key characteristics of PARSEC benchmarks. PARSEC workloads were chosen to cover different application domains, parallel models and runtime behaviors. Working sets that are 'unbounded' are large and have the additional qualitative property that there is significant application demand to make them even bigger. In practice they are typically only constrained by main memory size.

time, new applications are emerging that not only organize and catalog data on desktops and the Internet but also deliver improved visual experience [24].

These technology shifts have galvanized research in parallel architectures. Such research efforts rely on existing benchmark suites. However, as I described in Chapter 1 the existing suites [41,55,65,99] suffer from a number of limitations and are not adequate to evaluate future CMPs.

To address this problem, I created a publicly available benchmark suite called PARSEC. It includes not only a number of important RMS applications [24] but also several leading-edge applications from Princeton University, Stanford University, and the open-source domain. Since its release the suite has been downloaded thousands of times. More than 280 studies using PARSEC have already been published.

The work presented in this chapter was previously published in [10, 12].

2.2 The PARSEC Benchmark Suite

One of the goals of the PARSEC suite was to assemble a program selection that is large and diverse enough to be sufficiently representative for modern multiprocessors. It con-

sists of 13 workloads which were chosen from several application domains. PARSEC workloads were selected to include different combinations of parallel models, machine requirements and runtime behaviors. All benchmarks are written in C/C++ because of the continuing popularity of these languages in the near future. Table 2.1 presents a qualitative summary of their key characteristics.

PARSEC meets all the requirements outlined in Section 1.2.1:

- All applications are multithreaded. Many workloads even implement multiple different versions of the parallel algorithm which users can choose from.
- The PARSEC benchmark suite focuses on emerging workloads. The algorithms these programs implement are usually considered useful, but their computational demands are prohibitively high on contemporary platforms. As more powerful processors become available in the near future, they are likely to proliferate rapidly.
- The workloads are diverse and were chosen from many different areas such as computer vision, media processing, computational finance, enterprise servers and animation physics. PARSEC is more diverse than previous popular parallel benchmarks [9].
- Each of the applications chosen represents the state-of-art technique in its area. All workloads were developed in cooperation with experts from the respective application area. Some of the programs are modified versions of the research prototypes that were used to develop the implemented main algorithm.
- PARSEC supports computer architecture research in a number of ways. The most important one is that for each workload six input sets with different properties are defined. Three of these input sets are suitable for microarchitectural simulation. The different types of input sets are explained in more detail in Section 2.2.2.

2.2.1 Workloads

The PARSEC benchmark suite is composed of 10 applications and 3 kernels which represent common desktop and server programs. The workloads were chosen from different application domains such as computational finance, computer vision, real-time animation or media processing. The decision for the inclusion of a workload in the suite was based

Program	Problem Size	Instructions (Billions)				Synchronization Primitives		
		Total	FLOPS	Reads	Writes	Locks	Barriers	Conditions
blackscholes	65,536 options	4.90	2.32	1.51	0.79	0	8	0
bodytrack	4 frames, 4,000 particles	14.04	6.08	3.26	0.80	28,538	2,242	518
canneal	400,000 elements, 128 temperature steps	7.00	0.45	1.76	0.88	34	1,024	0
dedup	184 MB data	41.40	0.23	9.85	3.77	258,381	0	291
facesim	1 frame, 372,126 tetrahedra	30.46	17.17	9.91	4.23	14,566	0	3,327
ferret	256 queries, 34,973 images	25.90	6.58	7.65	1.99	534,866	0	1273
fluidanimate	5 frames, 300,000 particles	13.54	4.30	4.46	1.07	9,347,914	320	0
freqmine	990,000 transactions	33.22	0.08	11.19	5.23	990,025	0	0
raytrace	3 frames, $1,920 \times 1,080$ pixels	46.48	8.12	11.07	9.28	105	0	38
streamcluster	16,384 points per block, 1 block	22.15	16.49	4.26	0.06	183	129,584	115
swaptions	64 swaptions, 20,000 simulations	16.81	5.66	5.62	1.54	23	0	0
vips	1 image, $2,662 \times 5,500$ pixels	31.30	6.34	6.69	1.62	33,920	0	7,356
x264	128 frames, 640×360 pixels	14.42	7.37	3.88	1.16	16,974	0	1,101

Table 2.2: Breakdown of instructions and synchronization primitives of PARSEC workloads for input set `simlarge` on a system with 8 cores. All numbers are totals across all threads. Numbers for synchronization primitives also include primitives in system libraries. *Locks* and *Barriers* are all lock- and barrier-based synchronizations, *Conditions* are all waits on condition variables.

on the relevance of the type of problem it is solving, the distinctiveness of its characteristics as well as its overall novelty.

2.2.2 Input Sets

PARSEC defines six input sets for each benchmark:

`test` A very small input set to test the basic functionality of the program. The `test` input set gives no guarantees other than the benchmark will be executed. It should not be used for scientific studies.

`simdev` A very small input set which guarantees basic program behavior similar to the real behavior. It tries to preserve the code path of the real inputs as much as

possible. `Simdev` is intended for simulator test and development and should not be used for scientific studies.

`simsmall`, `simmedium` and `simlarge` Input sets of different sizes suitable for microarchitectural studies with simulators. The three simulator input sets vary in size but the general trend is that larger input sets contain bigger working sets and more parallelism.

`native` A large input set intended for native execution. It exceeds the computational demands which are generally considered feasible for simulation by orders of magnitude. From a scientific point of view, the `native` input set is the most interesting one because it resembles real program inputs most closely.

The three simulation input sets can be considered coarser approximations of the `native` input set which sacrifice accuracy for tractability. They were created by scaling down the size of the `native` input set. The methodology used for the size reduction and the involved trade-offs are described in Chapter 5. Table 2.2 shows a breakdown of instructions and synchronization primitives of the `simlarge` input set.

2.2.3 Threading Models

Parallel programming paradigms are a focus of computer science research due to their importance for making the large performance potential of CMPs more accessible. The PARSEC benchmark suite supports POSIX threads (pthreads), OpenMP and the Intel Threading Building Blocks (TBB). Table 2.3 summarizes which workloads support which threading models. Besides the constructs of these threading models, atomic instructions are also directly used by a few programs if synchronized low-latency data access is necessary.

POSIX threads [89] are one of the most commonly used threading standards to program contemporary shared-memory Unix machines. Pthreads requires programmers to handle all thread creation, management and synchronization issues themselves. It was officially finalized by IEEE in 1995 in section 1003.1c of the Portable Operating System Interface for Unix (POSIX) standard in an effort to harmonize and succeed the various threading standards that industry vendors had created themselves. This threading model is supported by all PARSEC workloads except `freqmine`.

Program	Pthreads	OpenMP	TBB
blackscholes	✓	✓	✓
bodytrack	✓	✓	✓
canneal	✓		
dedup	✓		
facesim	✓		
ferret	✓		
fluidanimate	✓		✓
freqmine		✓	
raytrace	✓		
streamcluster	✓		✓
swaptions	✓		✓
vips	✓		
x264	✓		

Table 2.3: Threading models supported by PARSEC.

OpenMP [71] is a compiler-based approach to program parallelization. To parallelize a program with OpenMP the programmer must annotate the source code with the OpenMP `#pragma omp` directives. The compiler performs the actual parallelization, and all details of the thread management and the synchronization are handled by the OpenMP runtime. The first version of the OpenMP API specification was released for Fortran in 1997 by the Architecture Review Board (ARB). OpenMP 1.0 for C/C++ followed the subsequent year. The standard keeps evolving, the latest version 3.0 was released in 2008. In the PARSEC benchmark suite OpenMP is supported by `blackscholes`, `bodytrack` and `freqmine`.

TBB is a high-level alternative to pthreads and similar threading libraries [40]. It can be used to parallelize C++ programs. The TBB library is a collection of C++ methods and templates which allow to express high-level, task-based parallelism that abstracts from details of the platform and the threading mechanism. The first version of the TBB library was released in 2006, which makes it one of the more recent threading models. The PARSEC benchmark suite has TBB support for five of its workloads: `blackscholes`, `bodytrack`, `fluidanimate`, `streamcluster` and `swaptions`.

Researchers that use the PARSEC benchmark suite for their work must be aware that the different versions of a workload that use the various threading methods can behave differently at runtime. Contreras et al. studied the TBB versions of the PARSEC workloads in more detail [21]. They conclude that the dynamic task handling approach of the

	Bit	Instruction	Data	Task
Exploitation	hardware	hardware	software	software
Granularity	bits	instructions	loops	functions
Dependencies	gates	registers	variables	control flow
Synchronization	clock signal	control logic	locks	conditions

Table 2.4: Levels of parallelism and their typical properties in practice. Only data and task parallelism are commonly exploited by software to take advantage of multiprocessors. Task parallelism is sometimes further subdivided into pipeline parallelism and 'natural' task parallelism to distinguish functions with a producer-consumer relationship from completely independent functions.

TBB runtime is effective at lower core counts, where it efficiently reduces load imbalance and improves scalability. However, with increasing core counts the overhead of the random task stealing algorithm becomes the dominant bottleneck. In current TBB implementations it can contribute up to 47% of the total per-core execution time on a 32-core system. Results like these demonstrate the importance of choosing a suitable threading model for performance experiments.

2.3 Pipeline Programming Model

The PARSEC benchmark suite is one of the first suites to include the pipeline model. Pipelining is a parallelization method that allows a program or system to execute in a decomposed fashion. It takes advantage of parallelism that exists on a function level. Table 2.4 gives a summary of the different levels of parallelism and how they are typically exploited. Information on the different types of data-parallel algorithms has been available for years [37], and existing benchmark suites cover data-parallel programs well [84, 96]. However, no comparable body of work exists for task parallelism, and the number of benchmark programs using pipelining to exploit parallelism on the task level is still limited. This section describes the pipeline parallelization model in more detail and how it is covered by the PARSEC benchmark suite.

A pipelined workload for multiprocessors breaks its work steps into units or *pipeline stages* and executes them concurrently on multiprocessors or multiple CPU cores. Each pipeline stage typically takes input from its input queue, which is the output queue of the previous stage, computes and then outputs to its output queue, which is the input queue

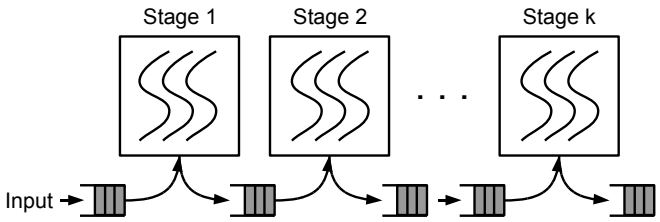

Figure 2.1: A typical linear pipeline with multiple concurrent stages. Pipeline stages have a producer - consumer relationship to each other and exchange data with queues.

of the next stage. Each stage can have one or more threads depending on specific designs. Figure 2.1 shows this relationship between stages and queues of the pipeline model.

2.3.1 Motivation for Pipelining

In practice there are three reasons why workloads are pipelined. First, pipelining can be used to simplify program engineering, especially for large-scale software development. Pipelining decomposes a problem into smaller, well-defined stages or pieces so that different design teams can develop different pipeline stages efficiently. As long as the interfaces between the stages are properly defined, little coordination is needed between the different development teams so that they can work independently from each other in practice. This typically results in improved software quality and lowered development cost due to simplification of the problem and specialization of the developers. This makes the pipeline model well suited for the development of large-scale software projects.

Second, the pipeline programming model can be used to take advantage of specialized hardware. Pipelined programs have clearly defined boundaries between stages, which make it easy to map them to different hardware and even different computer systems to achieve better hardware utilization.

Third, pipelining increases program throughput due to a higher degree of parallelism that can be exploited. The different pipeline stages of a workload can operate concurrently from each other, as long as enough input data is available. It can even result in fewer locks than alternative parallelization models [54] due to the serialization of data. By keeping data in memory and transferring it directly between the relevant processing elements, the pipeline model distributes the load and reduces the chance for bottlenecks. This has been a key motivation for the development of the stream programming

model [46], which can be thought of as a fine-grained form of the pipeline programming model.

2.3.2 Uses of the Pipeline Model

These properties of the pipeline model typically result in three uses in practice:

1. Pipelining as a hybrid model with data-parallel pipeline stages to increase concurrency
2. Pipelining to allow asynchronous I/O
3. Pipelining to model algorithmic dependencies

The first common use of the pipeline model is as a hybrid model that also exploits data parallelism. In that case the top-level structure of the program is a pipeline, but each pipeline stage is further parallelized so that it can process multiple work units concurrently. This program structure increases the overall concurrency and typically results in higher speedups.

The second use also aims to increase program performance by increasing concurrency, but it exploits parallelism between the CPUs and the I/O subsystem. This is done either by using special non-blocking system calls for I/O, which effectively moves that pipeline stage into the operating system, or by creating a dedicated pipeline stage that will handle blocking system calls so that the remainder of the program can continue to operate while the I/O thread waits for the operation to complete.

Lastly, pipelining is a method to decompose a complex program into simpler execution steps with clearly defined interfaces. This makes it popular to model algorithmic dependencies which are difficult to analyze and might even change dynamically at runtime. In that scenario the developer only needs to keep track of the dependencies and expose them to the operating system scheduler, which will pick and execute a job as soon as all its prerequisites are satisfied. The pipelines modeled in such a fashion can be complex graphs with multiple entry and exit points that have little in common with the linear pipeline structure that is typically used for pipelining.

2.3.3 Implementations

There are two ways to implement the pipeline model: fixed data and fixed code. The fixed data approach has a static mapping of data to threads. With this approach each thread applies all the pipeline stages to the work unit in the predefined sequence until the work unit has been completely processed. Each thread of a fixed data pipeline would typically take on a work unit from the program input and carry it through the entire program until no more work needs to be done for it, which means threads can potentially execute all of the parallelized program code but they will typically only see a small subset of the input data. Programs that implement fixed data pipelines are therefore also inherently data-parallel because it can easily happen that more than one thread is executing a function at any time.

The fixed code approach statically maps the program code of the pipeline stages to threads. Each thread executes only one stage throughout the program execution. Data is passed between threads in the order determined by the pipeline structure. For this reason each thread of a fixed code pipeline can typically only execute a small subset of the program code, but it can potentially see all work units throughout its lifetime. Pipeline stages do not have to be parallelized if no more than one thread is active per pipeline stage at any time, which makes this a straightforward approach to parallelize serial code.

Fixed Data Approach

The fixed data approach uses a static assignment of data to threads, each of which applies all pipeline stages to the data until completion of all tasks. The fixed data approach can be best thought of as a full replication of the original program, several instances of which are now executed concurrently and largely independently from each other. Programs that use the fixed data approach are highly concurrent and also implicitly exploit data parallelism. Due to this flexibility they are usually inherently load-balanced.

The key advantage of the fixed data approach is that it exploits data locality well. Because data does not have to be transferred between threads, the program can take full advantage of data locality once a work unit has been loaded into a cache. This assumes that threads do not migrate between CPUs, a property that is usually enforced by manually pinning threads to cores.

The key disadvantage is that it does not separate software modules to achieve a better division of labor for teamwork, simple asynchronous I/Os, or mapping to special hardware. The program will have to be debugged as a single unit. Asynchronous I/Os will need to be handled with concurrent threads. Typically, no fine-grained mapping to hardware is considered.

Another disadvantage of this approach is that the working set of the entire execution is proportional to the number of concurrent threads, since there is little data sharing among threads. If the working set exceeds the size of the low-level cache such as the level-two cache, this approach may cause many DRAM accesses due to cache misses. For the case that each thread contributes a relatively large working set, this approach may not be scalable to a large number of CPU cores.

Fixed Code Approach

The fixed code approach assigns a pipeline stage to each thread, which then exchange data as defined by the pipeline structure. This approach is very common because it allows the mapping of threads to different types of computational resources and even different systems.

The key advantage of this approach is its flexibility, which overcomes the disadvantages of the fixed data approach. As mentioned earlier, it allows fine-grained partitioning of software projects into well-defined and well-interfaced modules. It can limit the scope of asynchronous I/Os to one or a small number of software modules and yet achieves good performance. It allows engineers to consider fine-grained processing steps to fully take advantage of hardware. It can also reduce the aggregate working set size by taking advantage of efficient data sharing in a shared cache in a multiprocessor or a multicore CPU.

The main challenge of this approach is that each pipeline stage must use the right number of threads to create a load-balanced pipeline that takes full advantage of the target hardware because the throughput of the whole pipeline is determined by the rate of its slowest pipeline stage. In particular, pipeline stages can make progress at different rates on different systems, which makes it hard to find a fixed assignment of resources to stages for different hardware. A typical solution to this problem on shared-memory multiprocessor systems is to over-provision threads for pipeline stages so that it is guaranteed that enough cores can be assigned to each pipeline stage at any time. This solution

Workload	Parallelism			Dependency Modeling
	Pipeline	Data	I/O	
bodytrack		✓	✓	
dedup	✓	✓	✓	
ferret	✓	✓	✓	
x264	✓			✓

Table 2.5: The four workloads of PARSEC 2.1 which use the pipeline model. *Pipeline parallelism* in the table refers only to the decomposition of the computationally intensive parts of the program into separate stages and is different from the pipeline model as a form to structure the whole program (which includes stages to handle I/O).

delegates the task of finding the optimal assignment of cores to pipeline stages to the OS scheduler at runtime. However, this approach introduces additional scheduling overhead for the system.

Fixed code pipelines usually implement mechanisms to tolerate fluctuations of the progress rates of the pipeline stages, typically by adding a small amount of buffer space between stages that can hold a limited number of work units if the next stage is currently busy. This is done with synchronized queues on shared-memory machines or network buffers if two connected pipeline stages are on different systems. It is important to point out that this is only a mechanism to tolerate variations in the progress rates of the pipeline stages, buffer space does not increase the maximum possible throughput of a pipeline.

2.3.4 Pipelining in PARSEC

The PARSEC suite contains workloads implementing all the usage scenarios discussed in Section 2.3.2. Table 2.5 gives an overview of the four PARSEC workloads that use the pipeline model.

`Dedup` and `ferret` are server workloads which implement a typical linear pipeline with the fixed code approach (see Section 2.3.3). `X264` uses the pipeline model to model dependencies between frames. It constructs a complex pipeline at runtime based on its encoding decision in which each frame corresponds to a pipeline stage. The pipeline has the form of a directed, acyclical graph with multiple root nodes formed by the pipeline stages corresponding to the I frames. These frames can be encoded independently from other frames and thus do not depend on any input from other pipeline stages.

The `bodytrack` workload only uses pipelining to perform I/O asynchronously. It will be treated as a data-parallel program for the purposes of this study because it does not take advantage of pipeline parallelism in the computationally intensive parts. The remaining three pipelined workloads will be compared to the data-parallel programs in the PARSEC suite to determine whether the pipeline model has any influence on the characteristics.

2.4 Description of PARSEC Workloads

The following workloads are part of the PARSEC suite:

2.4.1 Blackscholes

The `blackscholes` application is an Intel RMS benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE) [15]

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

where V is an option on the underlying S with volatility σ at time t if the constant interest rate is r. There is no closed-form expression for the Black-Scholes equation and as such it must be computed numerically [39]. The `blackscholes` benchmark was chosen to represent the wide field of analytic PDE solvers in general and their application in computational finance in particular. The program is limited by the amount of floating-point calculations a processor can perform.

`Blackscholes` stores the portfolio with `numOptions` derivatives in array `OptionData`. The program includes file `optionData.txt` which provides the initialization and control reference values for 1,000 options which are stored in array `data_init`. The initialization data is replicated if necessary to obtain enough derivatives for the benchmark.

The program divides the portfolio into a number of work units equal to the number of threads and processes them concurrently. Each thread iterates through all derivatives in its contingent and calls function `BlkSchlsEqEuroNoDiv` for each of them to compute its price. If error checking was enabled at compile time it also compares the result with the reference price.

The inputs for `blackscholes` are sized as follows:

- `test`: 1 option
- `simdev`: 16 options
- `simsmall`: 4,096 options
- `simmedium`: 16,384 options
- `simlarge`: 65,536 options
- `native`: 10,000,000 options

The program writes the prices for all options to the output file `prices.txt`.

2.4.2 Bodytrack

The `bodytrack` computer vision application is an Intel RMS workload which tracks a 3D pose of a marker-less human body with multiple cameras through an image sequence [5, 23]. This is shown in Figure 2.2. `Bodytrack` employs an annealed particle filter to track the pose using edges and the foreground silhouette as image features, based on a 10 segment 3D kinematic tree body model. These two image features were chosen because they exhibit a high degree of invariance under a wide range of conditions and because they are easy to extract. An annealed particle filter was employed in order to be able to search high dimensional configuration spaces without having to rely on any assumptions of the tracked body such as the existence of markers or constrained movements. This benchmark was included due to the increasing significance of computer vision algorithms in areas such as video surveillance, character animation and computer interfaces.

For every frame set Z_t of the input videos at time step t, the `bodytrack` benchmark executes the following steps:

1. The image features of observation Z_t are extracted. The features will be used to compute the likelihood of a given pose in the annealed particle filter.
2. Every time step t the filter makes an annealing run through all M annealing layers, starting with layer $m = M$.

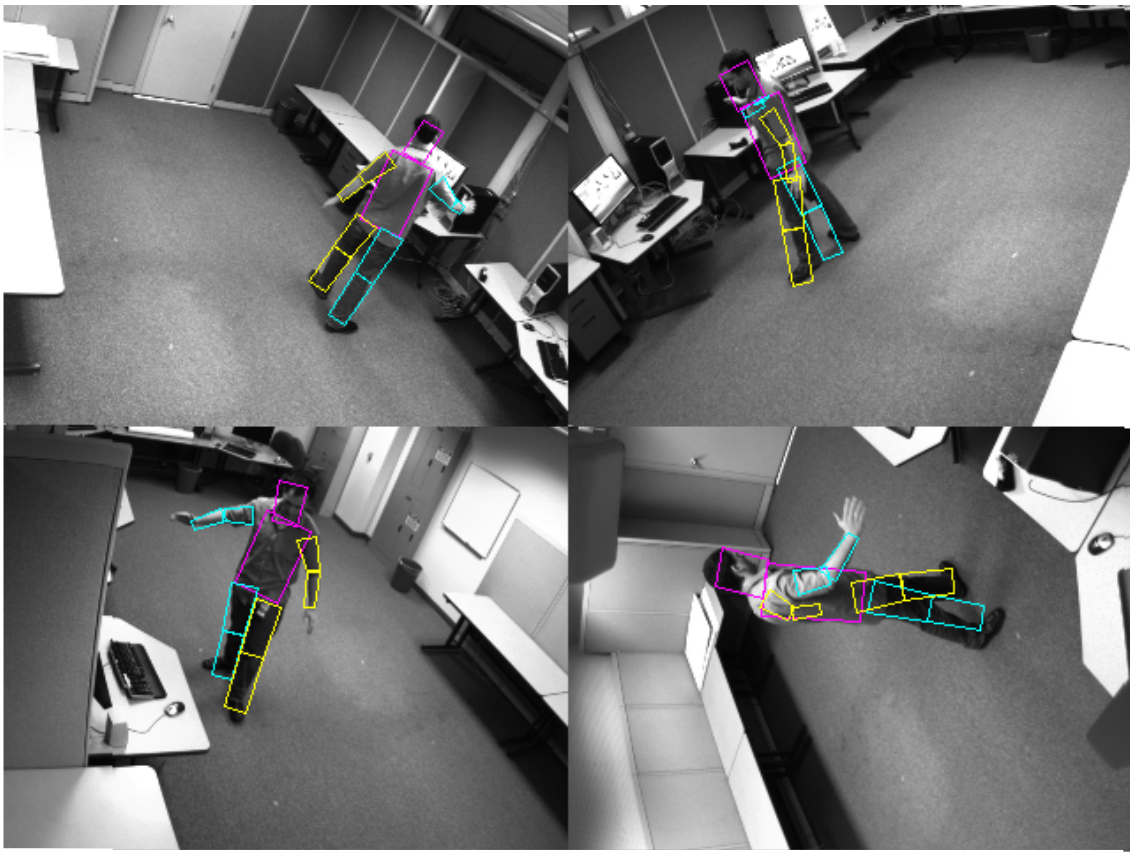

Figure 2.2: Output of the `bodytrack` benchmark. The program recognizes the body position of the person in the input videos and adds boxes to mark it for a human observer.

3. Each layer m uses a set of N unweighted particles which are the result of the previous filter update step to begin with.

$$S_{t,m} = \{(s_{t,m}^{(1)})...(s_{t,m}^{(N)})\}.$$

   Each particle $s_{t,m}^{(i)}$ is an instance of the multi-variate model configuration X which encodes the location and state of the tracked body.

4. Each particle $s_{t,m}^{(i)}$ is then assigned a weight $\pi_{t,m}^{(i)}$ by using weighting function $\omega(Z_t, X)$ corresponding to the likelihood of X given the image features in Z_t scaled by an annealing level factor:

$$\pi_{t,m}^{(i)} \propto \omega(Z_t, s_{t,m}^{(i)}).$$

The weights are normalized so that $\sum_{i=1}^{N} \pi_{t,m}^{(i)} = 1$. The result is the weighted particle set

$$S_{t,m}^{\pi} = \{(s_{t,m}^{(1)}, \pi_{t,m}^{(1)}) ... (s_{t,m}^{(N)}, \pi_{t,m}^{(N)})\}.$$

5. N particles are randomly drawn from set $S_{t,m}^{\pi}$ with a probability equal to their weight $\pi_{t,m}^{(i)}$ to obtain the temporary weighted particle set

$$\bar{S}_{t,m}^{\pi} = \{(\bar{s}_{t,m}^{(1)}, \pi_{t,m}^{(1)}) ... (\bar{s}_{t,m}^{(N)}, \pi_{t,m}^{(N)})\}.$$

Each particle $\bar{s}_{t,m}^{(i)}$ is then used to produce particle

$$s_{t,m-1}^{(i)} = \bar{s}_{t,m}^{(i)} + B_m$$

where B_m is a multi-variate Gaussian random variable. The result is particle set $S_{t,m-1}^{\pi}$ which is used to initialize layer $m-1$.

6. The process is repeated until all layers have been processed and the final particle set $S_{t,0}^{\pi}$ has been computed.

7. $S_{t,0}^{\pi}$ is used to compute the estimated model configuration χ_t for time step t by calculating the weighted average of all configuration instances:

$$\chi_t = \sum_{i=1}^{N} s_{t,0}^{(i)} \pi_{t,0}^{(i)}.$$

8. The set $S_{t+1,M}$ is then produced from $S_{t,0}^{\pi}$ using

$$s_{t+1,M}^{(i)} = s_{t,0}^{(i)} + B_0.$$

In the subsequent time step $t+1$ the set $S_{t+1,M}$ is used to initialize layer M.

The likelihood $\omega(Z_t, s_{t,m}^{(i)})$ which is used to determine the particle weights $\pi_{t,m}^{(i)}$ is computed by projecting the geometry of the human body model into the image observations Z_t for each camera and determining the error based on the image features. The likelihood

is a measure of the 3D body model alignment with the foreground and edges in the images. The body model consists of conic cylinders to represent 10 body parts 2 for each limb plus the torso and the head. Each cylinder is represented by a length and a radius for each end. The body parts are assembled into a kinematic tree based upon the joint angles. Each particle represents the set of joint angles plus a global translation. To evaluate the likelihood of a given particle, the geometry of the body model is first built in 3D space given the angles and translation. Next, each 3D body part is projected onto each of the 2D images as a quadrilateral. A likelihood value is then computed based on the two image features the foreground map and the edge distance map. To compute the foreground term, samples are taken within the interior of each 2D body part projection and compared with the binary foreground map images. Samples that correspond to foreground increase the likelihood while samples that correspond to background are penalized. The edge map gives a measure of the distance from an edge in the image - values closer to an edge have a higher value. To compute the edge term samples are taken along the axis-parallel edges of each 2D body part projection and the edge map values at each sample are summed together. In this way, samples that are closer to edges in the images increase the likelihood while samples farther from edges are penalized.

`Bodytrack` has a persistent thread pool which is implemented in class `WorkPool-Pthread`. Input images that form the individual observations are loaded using asynchronous I/O so that disk I/O and computations are overlapping. The main thread executes the program and sends a task to the thread pool with method `SignalCmd` whenever it reaches a parallel kernel. It resumes execution of the program as soon as it receives the result from the worker threads. Possible tasks are encoded by enumeration `threadCommands` in class `WorkPoolPthread`. The program has three parallel kernels:

Edge detection (Step 1) `Bodytrack` employs a gradient based edge detection mask to find edges. The result is compared against a threshold to eliminate spurious edges. Edge detection is implemented in function `GradientMagThreshold`. The output of this kernel will be further refined before it is used to compute the particle weights.

Edge smoothing (Step 1) A separable Gaussian filter of size 7×7 pixels is used to smooth the edges in function `GaussianBlur`. The result is remapped between 0 and 1 to produce a pixel map in which the value of each pixel is related to its dis-

tance from an edge. The kernel has two parallel phases, one to filter image rows and one to filter image columns.

Calculate particle weights (Step 4) This kernel evaluates the foreground silhouette and the image edges produced earlier to compute the weights for the particles. This kernel is executed once for every annealing layer during every time step, making it the computationally most intensive part of the body tracker.

Particle resampling (Step 5) This kernel resamples particles by adding normally distributed random noise to them, thereby effectively creating a new set of particles. This function is implemented in `GenerateNewParticles`. The random number generator used for the task is given by class `RandomGenerator`.

The parallel kernels use tickets to distribute the work among threads balance the load dynamically. The ticketing mechanism is implemented in class `TicketDispenser` and behaves like a shared counter.

The inputs for `bodytrack` are defined as follows:

- `test`: 4 cameras, 1 frame, 5 particles, 1 annealing layer
- `simdev`: 4 cameras, 1 frame, 100 particles, 3 annealing layers
- `simsmall`: 4 cameras, 1 frame, 1,000 particles, 5 annealing layers
- `simmedium`: 4 cameras, 2 frames, 2,000 particles, 5 annealing layers
- `simlarge`: 4 cameras, 4 frames, 4,000 particles, 5 annealing layers
- `native`: 4 cameras, 261 frames, 4,000 particles, 5 annealing layers

The result of the computations is written to the output file `poses.txt`. The program also creates output images that mark the exact location of the recognized body as shown in Figure 2.2.

2.4.3 Canneal

This kernel uses cache-aware simulated annealing (SA) to minimize the routing cost of a chip design [6]. SA is a common method to approximate the global optimum in a large search space. `Canneal` pseudo-randomly picks pairs of elements and tries to swap them. To increase data reuse, the algorithm discards only one element during each iteration which effectively reduces cache capacity misses. The SA method accepts swaps which increase the routing cost with a certain probability to make an escape from local optima possible. This probability continuously decreases during runtime to allow the design to converge. For benchmarking version the number of temperature steps has been fixed to keep the amount of work constant. The program was included in the PARSEC program selection to represent engineering workloads, for the fine-grained parallelism with its lock-free synchronization techniques and due to its pseudo-random worst-case memory access pattern.

`Canneal` uses a very aggressive synchronization strategy that is based on data race recovery instead of avoidance. Pointers to the elements are dereferenced and swapped atomically, but no locks are held while a potential swap is evaluated. This can cause disadvantageous swaps if one of the relevant elements has been replaced by another thread during that time. This equals a higher effective probability to accept swaps which increase the routing cost, and the SA method automatically recovers from it. The swap operation employs lock-free synchronization which is implemented with atomic instructions. An alternative implementation which relied on conventional locks turned out to be too inefficient due to excessive locking overhead. The synchronization routines with the atomic instructions are taken from the BSD kernel. Support for most new architectures can be added easily by copying the correct header file from the BSD kernel sources.

The annealing algorithm is implemented in the `Run` function of the `annealer_thread` class. Each thread uses the function `get_random_element` to pseudo-randomly pick one new netlist element per iteration with a Mersenne Twister [63]. `calculate_delta_routing_cost` is called to compute the change of the total routing cost if the two elements are swapped. `accept_move` evaluates the change in cost and the current temperature and decides whether the change is to be committed. Finally, accepted swaps are executed by calling `swap_locations`.

`Canneal` implements an `AtomicPtr` class which encapsulates a shared pointer to the location of a netlist element. The pointer is atomically accessed and modified with the

`Get` and `Set` functions offered by the class. A special `Swap` member function executes an atomic swap of two encapsulated pointers. If an access is currently in progress the functions spin until the operation could be completed. The implementation of `Swap` imposes a partial order to avoid deadlocks by processing the pointer at the lower memory location first.

The following inputs are provided for `canneal`:

- `test`: 5 swaps per temperature step, 100° start temperature, 10 netlist elements, 1 temperature step
- `simdev`: 100 swaps per temperature step, 300° start temperature, 100 netlist elements, 2 temperature steps
- `simsmall`: 10,000 swaps per temperature step, $2,000$° start temperature, 100,000 netlist elements, 32 temperature steps
- `simmedium`: 15,000 swaps per temperature step, $2,000$° start temperature, 200,000 netlist elements, 64 temperature steps
- `simlarge`: 15,000 swaps per temperature step, $2,000$° start temperature, 400,000 netlist elements, 128 temperature steps
- `native`: 15,000 swaps per temperature step, $2,000$° start temperature, 2,500,000 netlist elements, 6,000 temperature steps

The program writes the final routing cost to the console.

2.4.4 Dedup

The `dedup` kernel compresses a data stream with a combination of global compression and local compression in order to achieve high compression ratios. Such a compression is called *deduplication*. The reason for the inclusion of this kernel is that deduplication has become a mainstream method to reduce storage footprints for new-generation backup storage systems [76] and to compress communication data for new-generation bandwidth optimized networking appliances [85].

The kernel uses the pipeline programming model to parallelize the compression to mimic real-world implementations. There are five pipeline stages, the intermediate three

of which are parallel. In the first stage, `dedup` reads the input stream and breaks it up into coarse-grained chunks to get independent work units for the threads. The second stage anchors each chunk into fine-grained small segments with rolling fingerprinting [16, 61]. The third pipeline stage computes a hash value for each data segment. The fourth stage compresses each data segment with the Ziv-Lempel algorithm and builds a global hash table that maps hash values to data. The final stage assembles the deduplicated output stream consisting of hash values and compressed data segments.

Anchoring is a method which identifies brief sequences in a data stream that are identical with sufficiently high probability. It uses rolling fingerprints [45, 77] to segment data based on its contents. The data is then broken up into two separate blocks at the determined location. This method ensures that fragmenting a data stream is unlikely to obscure duplicate sequences since duplicates are identified on a block basis.

`Dedup` uses a separate thread pool for each parallel pipeline stage. Each thread pool should at least have a number of threads equal to the number of available cores to allow the system to fully work on any stage should the need arise. The operating system scheduler is responsible for a thread schedule which will maximize the overall throughput of the pipeline. In order to avoid lock contention, the number of queues is scaled with the number of threads, with a small group of threads sharing an input and output queue at a time.

`Dedup` employs the following five kernels, one for each pipeline stage:

Coarse-grained fragmentation This serial kernel takes the input stream and breaks it up into work units which can be processed independently from each other by the parallel pipeline stages of `dedup`. It is implemented in function `Fragment`. First, the kernel reads the input file from disk. It then determines the locations where the data is to be split up by jumping a fixed length in the buffer for each chunk. The resulting data blocks are enqueued in order to be further refined by the subsequent stage.

Fine-grained fragmentation This parallel kernel uses Rabin fingerprints to break a coarse-grained data chunk up into fine-grained fragments. It scans each input block starting from the beginning. An anchor is found if the lowest 12 bits of the Rabin hash sum are 0. The data is then split up at the location of the anchor. On average, this produces blocks of size $2^{12}/8 = 512$ bytes. The fine-grained data blocks are

sent to the subsequent pipeline stage to compute their checksum. This kernel is implemented in function `FragmentRefine`.

Hash computation To uniquely identify a fine-grained data block, this parallel kernel computes the SHA1 checksum of each chunk and checks for duplicate blocks with the use of a global database. It is implemented in function `Deduplicate`. A hash table which is indexed with the SHA1 sum serves as the database. Each bucket of the hash table is associated with an independent lock in order to synchronize accesses. The large number of buckets and therefore locks makes the probability of lock contention very low in practice.

Once the SHA1 sum of a data block is available, the kernel checks whether a corresponding entry already exists in the database. If no entry could be found, the data block is added to the hash table and sent to the compression stage. If an entry already exists the block is classified as a duplicate. The compression stage is omitted and the block is sent directly to the pipeline stage which assembles the output stream.

Compression This kernel compresses data blocks in parallel. It is implemented in function `Compress`. Once the compressed image of a data block is available it is added to the database and the corresponding data block is sent to the next pipeline stage. Every data block is compressed only once because the previous stage does not send duplicates to the compression stage.

Assemble output stream This serial kernel reorders the data blocks and produces a compressed output stream. It is implemented in the `Reorder` function. The stages which fragment the input stream into fine-grained data blocks add sequence numbers to allow a reconstruction of the original order. Because data fragmentation occurs in two different pipeline stages, two levels of sequence numbers have to be considered - one for each granularity level. `Reorder` uses a search tree for the first level and a heap for the second level. The search tree allows rapid searches for the correct heap corresponding to the current first-level sequence number. For second-level sequence numbers only the minimum has to be found and hence a heap is used.

Once the next data block in the sequence becomes available it is removed from the reordering structures. If it has not been written to the output stream yet, its compressed image is emitted. Otherwise it is a duplicate and only its SHA1 signature is written as a placeholder. The kernel uses the global hash table to keep track of the output status of each data block.

Each input for `dedup` is an archive which contains a selection of files. The archives have the following sizes:

- `test`: 10 KB
- `simdev`: 1.1 MB
- `simsmall`: 10 MB
- `simmedium`: 31 MB
- `simlarge`: 184 MB
- `native`: 672 MB

`Dedup` writes the compressed data stream to an output file which can then be given to its decompression utility to restore the original input data.

2.4.5 Facesim

This Intel RMS application was originally developed by Stanford University. It takes a model of a human face and a time sequence of muscle activations and computes a visually realistic animation of the modeled face by simulating the underlying physics [81, 88]. The goal is to create a visually realistic result. Two examples of a fully rendered face that were computed by `facesim` can be seen in Figure 2.3. Certain effects such as inertial movements would have only a small visible effect and are not simulated [38]. The workload was included in the benchmark suite because an increasing number of computer games and other forms of animation employ physical simulation to create more realistic virtual environment. Human faces in particular are observed with more attention from users than other details of a virtual world, making their realistic presentation a key element for animations.

Figure 2.3: Two fully rendered output frames of the `facesim` benchmark. The workload is able to compute a realistically looking face as a response to factors such as bone and muscle configuration or external forces.

The parallelization uses a static partitioning of the mesh. Data that spans nodes belonging to more than one partition is replicated. Every time step the partitions process all elements that contain at least one node owned by the particle, but only results for nodes which are owned by the partition are written.

The iteration which computes the state of the face mesh at the end of each iteration is implemented in function `Advance_One_Time_Step_Quasistatic`. `Facesim` employs the fork-join model to process computationally intensive tasks in parallel. It uses the following three parallel kernels for its computations:

Update state This kernel uses the Newton-Raphson method to solve the nonlinear system of equations in order to find the steady state of the simulated mesh. This quasi-static scheme achieves speedups of one to two orders of magnitudes over explicit schemes by ignoring inertial effects. It is not suitable for the simulation of less constrained phenomena such as ballistic motion, but it is sufficiently accurate

to simulate effects such as flesh deformation where the material is heavily influenced by contact, collision and self-collision and inertial effects only have a minor impact on the state.

In each Newton-Raphson iteration, the kernel reduces the nonlinear system of equations to a linear system which is guaranteed to be positive definite and symmetric. These two properties allow the use of a fast conjugate gradient solver later on. One iteration step is computed by function `Update_Position_Based_State`. The matrix of the linear system is sparse and can hence be stored in two one-dimensional arrays - `dX_full` and `R_full`. The matrix is the sum of the contribution of each tetrahedron of the face mesh.

Add forces This module computes the velocity-independent forces acting on the simulation mesh. After the matrix of the linear system with the position-independent state has been computed by the previous kernel, the right-hand side of that system has to be calculated. The kernel does this by iterating over all tetrahedra of the mesh, reading the positions of the vertices and computing the force contribution to each of the four nodes.

Conjugate gradient This kernel uses the conjugate gradient algorithm to solve the linear equation system assembled by the previous two modules. The two arrays `dX_full` and `R_full` which store the sparse matrix are sequentially accessed and matrix-vector multiplication is employed to solve the system.

The inputs of `facesim` all use the same face mesh. Scaling down the resolution of the mesh to create more tractable input sizes is impractical. A reduction of the number of elements in the model would result in under-resolution of the muscle action and cause problems for collision detection [38]. The inputs for `facesim` are defined as follows:

- `test`: Print out help message.
- `simdev`: 80,598 particles, 372,126 tetrahedra, 1 frame
- `simsmall`: Same as `simdev`
- `simmedium`: Same as `simdev`
- `simlarge`: Same as `simdev`

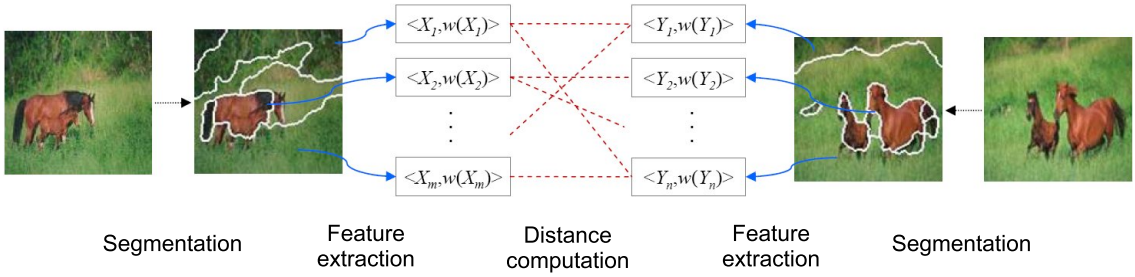

Figure 2.4: Algorithm of the `ferret` benchmark. Images are first segmented and a feature vector is computed for each segment. To compute the final similarity score the pair-wise distances between all segments of two images are calculated.

- `native`: Same as `simdev`, but with 100 frames

The benchmark writes the final state of the face mesh to several files whose names start with `deformable_object`.

2.4.6 Ferret

This application is based on the Ferret toolkit which is used for content-based similarity search of feature-rich data such as audio, images, video, 3D shapes and so on [59]. The reason for the inclusion in the benchmark is that it represents emerging next-generation desktop and Internet search engines for non-text document data types. For the benchmark the Ferret toolkit was configured for image similarity search. Ferret is parallelized using the pipeline model with six stages. The first and the last stage are for input and output. The middle four stages are for query image segmentation, feature extraction, indexing of candidate sets with multi-probe Locality Sensitive Hashing (LSH) [60] and ranking. Each stage has its own thread pool and the basic work unit of the pipeline is a query image.

Segmentation is the process of decomposing an image into separate areas which display different objects. The rationale behind this step is that in many cases only parts of an image are of interest, such as the foreground. Segmentation allows the subsequent stages to assign a higher weight to image parts which are considered relevant and seem to belong together. After segmentation, `ferret` extracts a feature vector from every segment. A feature vector is a multi-dimensional mathematical description of the segment contents. It encodes fundamental properties such as color, shape and area. Once the feature vectors are known, the indexing stage can query the image database to obtain a

candidate set of images. The database is organized as a set of hash tables which are indexed with multi-probe LSH [60]. This method uses hash functions which map similar feature vectors to the same hash bucket with high probability. Because the number of hash buckets is very high, multi-probe LSH first derives a probing sequence which considers the success probabilities for finding a candidate image in a bucket. It then employs a step-wise approach which indexes buckets with a higher success probability first. After a candidate set of images has been obtained by the indexing stage, it is sent to the ranking stage which computes a detailed similarity estimate and orders the images according to their calculated rank. The similarity estimate is derived by analyzing and weighing the pair-wise distances between the segments of the query image and the candidate images. The underlying metric employed is the Earth Mover's Distance (EMD) [79]. For two images X and Y, it is defined as

$$\mathrm{EMD}(X,Y) = \min \sum_i \sum_j f_{ij} d(X_i, Y_j)$$

where X_i and Y_j denote segments of X and Y and f_{ij} is the extent to which X_i is matched to Y_j. The fundamental steps of the algorithm can be seen in Figure 2.4.

The first and the last pipeline stage of `ferret` are serial. The remaining four modules are parallel:

Image segmentation This kernel uses computer vision techniques to break an image up into non-overlapping segments. The pipeline stage is implemented in function `t_seg`, which calls `image_segment` for every image. This function uses statistical region merging (SRM) [67] to segment the image. This method organizes the pixels of an image in sets, starting with a fine-granular decomposition. It repeatedly merges them until the final segmentation has been reached.

Feature extraction This module computes a 14-dimensional feature vector for each image segment. The features extracted are the bounding box of the segment (5 dimensions) and its color moments (9 dimensions). A bounding box is the minimum axis-aligned rectangle which includes the segment. Color moments is a compact representation of the color distribution. It is conceptually similar to a histogram but uses fewer dimensions. Segments are assigned a weight which is proportional

to the square root of its size. This stage is implemented in function `t_extract`. It calls `image_extract_helper` to compute the feature vectors for every image.

Indexing The indexing stage queries the image database to obtain no more than twice the number of images which are allowed to appear in the final ranking. This stage is implemented in function `t_vec`. `Ferret` manages image data in tables which have type `cass_table_t`. Tables can be queried with function `cass_table_query`. The indexing stage uses this function to access the database in order to generate the candidate set of type `cass_result_t` for the current query image. Indexing employs LSH for the probing which is implemented in function `LSH_query`.

Ranking This module performs a detailed similarity computation. From the candidate set obtained by the indexing stage it chooses the final set of images which are most similar to the query image and ranks them. The ranking stage is implemented in function `t_rank`. It employs `cass_table_query` to analyze the candidate set and to compute the final ranking with EMD. The type of query that `cass_table_query` is to perform can be described with a structure of type `cass_query_t`.

The number of query images determine the amount of parallelism. The working set size is dominated by the size of the image database. The inputs for `ferret` are sized as follows:

- `test`: 1 image queries, database with 1 image, find top 1 image
- `simdev`: 4 image queries, database with 100 images, find top 5 images
- `simsmall`: 16 image queries, database with 3,544 images, find top 10 images
- `simmedium`: 64 image queries, database with 13,787 images, find top 10 images
- `simlarge`: 256 image queries, database with 34,973 images, find top 10 images
- `native`: 3,500 image queries, database with 59,695 images, find top 50 images

The workload prints the result of its computations to the console.

Figure 2.5: An example for the use of particle effects in contemporary video games. The figure shows a screenshot of *Tom Clancy's Ghost Recon Advanced Warfighter* with (left) and without (right) particle effects.

2.4.7 Fluidanimate

This Intel RMS application uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes [64]. Its output can be visualized by detecting and rendering the surface of the fluid. The force density fields are derived directly from the Navier-Stokes equation. `Fluidanimate` uses special-purpose kernels to increase stability and speed. `Fluidanimate` was included in the PARSEC benchmark suite because of the increasing significance of physics simulations for computer games and other forms of real-time animations. An example for particle effects in video games can be seen in Figure 2.5.

A simplified version of the Navier-Stokes equation for incompressible fluids [75] which formulates conservation of momentum is

$$\rho(\frac{\partial v}{\partial t} + v \cdot \nabla v) = -\nabla p + \rho g + \mu \nabla^2 v$$

where v is a velocity field, ρ a density field, p a pressure field, g an external force density field and μ the viscosity of the fluid. The SPH method uses particles to model the state of the fluid at discrete locations and interpolates intermediate values with radial symmetrical smoothing kernels. An advantage of this method is the automatic conservation of mass due to a constant number of particles, but it alone does not guarantee certain physical principals such as symmetry of forces which have to be enforced separately. The SPH algorithm derives a scalar quantity A_S at location r by a weighted sum of all particles:

$$A_S(r) = \sum_j m_j \frac{A_j}{\rho_j} W(r - r_j, h).$$

In the equation, j iterates over all particles, m_j is the mass of particle j, r_j its position, ρ_j the density at its location and A_j the respective field quantity. $W(r - r_j, h)$ is the smoothing kernel to use for the interpolation with core radius h. Smoothing kernels are employed in order to make the SPH method stable and accurate. Because each particle i represents a volume with constant mass m_i, the density ρ_i appears in the equation and has to be recomputed every time step. The density at a location r can be calculated by substituting A with ρ_S in the previous equation:

$$\rho_S(r) = \sum_j m_j W(r - r_j, h)$$

.

Applying the SPH interpolation equation to the pressure term $-\nabla p$ and the viscosity term $\mu\nabla^2$ of the Navier-Stokes equation yields the equations for the pressure and viscosity forces, but in order to solve the force symmetry problems of the SPH method, `fluidanimate` employs slightly modified formulas:

$$f_i^{\text{pressure}} = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(r_i - r_j, h)$$

$$f_i^{\text{viscosity}} = \mu \sum_j m_j \frac{v_i - v_j}{\rho_j} \nabla^2 W(r_i - r_j, h)$$

Stability, accuracy and speed of `fluidanimate` are highly dependent on its smoothing kernels. In all cases except the pressure and viscosity computations the program uses the following kernel:

$$W_{\text{poly6}}(r,h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & 0 \le r \le h \\ 0 & \text{else} \end{cases}$$

One feature of this kernel is that the distance r only appears squared. The computation of square roots is thus not necessary to evaluate it. For pressure computations, `fluidanimate` uses Desbrun's spiky kernel W_{spiky} [22] and $W_{\text{viscosity}}$ for viscosity forces:

$$W_{\text{spiky}}(r,h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & 0 \le r \le h \\ 0 & \text{else} \end{cases}$$

$$W_{\text{viscosity}}(r,h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & 0 \le r \le h \\ 0 & \text{else} \end{cases}$$

The scene geometry employed by `fluidanimate` is a box in which the fluid resides. All collisions are handled by adding forces in order to change the direction of movement of the involved particles instead of modifying the velocity directly. The workload uses Verlet integration [93] to update the position of the particles. This method offers greater numerical stability that simpler approaches because it does not store the velocity of the particles. Instead it keeps track of the current and the last position. The velocity can thus be implicitly calculated by dividing the distance between the two positions by the length of the time step used by the simulator.

Every time step, `fluidanimate` executes five kernels, the first two of which were further broken up into several smaller steps:

Rebuild spatial index Because the smoothing kernels $W(r - r_j, h)$ have finite support h, particles can only interact with each other up to the distance h. The program uses a spatial indexing structure in order to exploit proximity information and limit the number of particles which have to be evaluated. Functions `ClearParticles` and `RebuildGrid` build this acceleration structure which is used by the subsequent steps.

Compute densities This kernel estimates the fluid density at the position of each particle by analyzing how closely particles are packed in its neighborhood. In a region in which particles are packed together more closely, the density will be higher. This kernel has 3 phases which are implemented in the functions `InitDensitiesAndForces`, `ComputeDensities` and `ComputeDensities2`.

Compute forces Once the densities are known, they can be used to compute the forces. This step happens in function `ComputeForces`. The kernel evaluates pressure, viscosity and also gravity as the only external influence. Collisions between particles are handled implicitly during this step, too.

Handle collisions with scene geometry The next kernel updates the forces in order to handle collisions of particles with the scene geometry. This step is implemented in function `ProcessCollisions`.

Update positions of particles Finally, the forces can be used to calculate the acceleration of each particle and update its position. `Fluidanimate` uses a Verlet integrator [93] for these computations which is implemented in function `AdvanceParticles`.

The inputs for `fluidanimate` are sized as follows:

- `test`: 5,000 particles, 1 frame
- `simdev`: 15,000 particles, 3 frames
- `simsmall`: 35,000 particles, 5 frames
- `simmedium`: 100,000 particles, 5 frames
- `simlarge`: 300,000 particles, 5 frames
- `native`: 500,000 particles, 500 frames

The benchmark writes the state of the fluid at the end of the computations to a user-determined output file. For all standard inputs provided by PARSEC this file is named `out.fluid`.

2.4.8 Freqmine

The `freqmine` application employs an array-based version of the FP-growth (Frequent Pattern-growth) method [30] for Frequent Itemset Mining (FIMI). It is an Intel RMS benchmark which was originally developed by Concordia University. FIMI is the basis of Association Rule Mining (ARM), a very common data mining problem which is relevant for areas such as protein sequences, market data or log analysis. The serial program this benchmark is based on won the FIMI'03 best implementation award for its efficiency. `Freqmine` was included in the PARSEC benchmark suite because of the increasing demand for data mining techniques which is driven by the rapid growth of the volume of stored information.

FP-growth stores all relevant frequency information of the transaction database in a compact data structure called FP-tree (Frequent Pattern-tree) [32]. An FP-tree is composed of three parts: First, a prefix tree encodes the transaction data such that each branch represents a frequent itemset. The nodes along the branches are stored in decreasing order of frequency of the corresponding item. The prefix tree is a more compact representation of the transaction database because overlapping itemsets share prefixes of the corresponding branches. The second component of the FP-tree is a header table which stores the number of occurrences of each item in decreasing order of frequency. Each entry is also associated with a pointer to a node of the FP-tree. All nodes which are associated with the same item are linked to a list. The list can be traversed by looking up the corresponding item in the header table and following the links to the end. Each node furthermore contains a counter that encodes how often the represented itemset as seen from the root to the current node occurs in the transaction database. The third component of the FP-tree is a lookup table which stores the frequencies of all 2-itemsets. A row in the lookup table gives all occurrences of items in itemsets which end with the associated item. This information can be used during the mining phase to omit certain FP-tree scans and is the major improvement of the implemented algorithm. The lookup table is especially effective if the dataset is sparse which is usually the case. The FP-trees are then very big due to the fact that only few prefixes are shared. In that case tree traversals are more expensive, and the benefit from being able to omit them is greater. The initial FP-tree can be constructed with only two scans of the original database, the first one to construct the header table and the second one to compute the remaining parts of the FP-tree.

In order to mine the data for frequent itemsets, the FP-growth method traverses the FP-tree data structure and recursively constructs new FP-trees until the complete set of frequent itemsets is generated. To construct a new FP-tree $T_{X\cup\{i\}}$ for an item i in the header of an existing FP-tree T_X, the algorithm first obtains a new pattern base from the lookup table. The base is used to initialize the header of the new tree $T_{X\cup\{i\}}$. Starting from item i in the header table of the existing FP-tree T_X, the algorithm then traverses the associated linked list of all item occurrences. The patterns associated with the visited branches are then inserted into the new FP-tree $T_{X\cup\{i\}}$. The resulting FP-tree is less bushy because it was constructed from fewer itemsets. The recursion terminates when an FP-tree was built which has only one path. The properties of the algorithm guarantee that this is a frequent itemset.

`Freqmine` has been parallelized with OpenMP. It employs three parallel kernels:

Build FP-tree header This kernel scans the transaction database and counts the number of occurrences of each item. It performs the first of two database scans necessary to construct the FP-tree. The result of this operation is the header table for the FP-tree which contains the item frequency information. This kernel has one parallelized loop and is implemented in function `scan1_DB`.

Construct prefix tree The next kernel builds the initial tree structure of the FP-tree. It performs the second and final scan of the transaction database necessary to build the data structures which will be used for the actual mining operation. The kernel has four parallelized loops. It is implemented in function `scan2_DB` which contains two of them. The remaining two loops are in its helper function `database_tiling`.

Mine data The last kernel uses the data structures previously computed and mines them to recursively obtain the frequent itemset information. It is an improved version of the conventional FP-growth method [32]. This module has similarities with the previous two kernels which construct the initial FP-tree because it builds a new FP-tree for every recursion step.

The module is implemented in function `FP_growth_first`. It first derives the initial lookup table from the current FP-tree by calling `first_transform_FPTree_into_FPArray`. This function executes the first of two parallelized loops. After that the second parallelized loop is executed in which the recursive function `FP_growth`

is called. It is the equivalent of `FP_growth_first`. Each thread calls `FP_growth` independently so that a number of recursions up to the number of threads can be active.

The inputs for `freqmine` are defined as follows:

- `test`: Database with 3 synthetic transactions, minimum support 1.
- `simdev`: Database with 1,000 synthetic transactions, minimum support 3.
- `simsmall`: Database with 250,000 anonymized click streams from a Hungarian online news portal, minimum support 220.
- `simmedium`: Same as `simsmall` but with 500,000 click streams, minimum support 410.
- `simlarge`: Same as `simsmall` but with 990,000 click streams, minimum support 790.
- `native`: Database composed of spidered collection of 250,000 web HTML documents [57], minimum support 11,000.

`Freqmine` outputs the results of its computations to the console.

2.4.9 Raytrace

The `raytrace` application is an Intel RMS workload which renders an animated 3D scene. Ray tracing is a technique that generates a visually realistic image by tracing the path of light through a scene [94]. Its major advantage over alternative rendering methods is its ability to create photorealistic images at the expense of higher computational requirements because certain effects such as reflections and shadows that are difficult to incorporate into other rendering methods are a natural byproduct of its algorithm. Ray tracing leverages the physical property that the path of light is always reversible to reduce the computational requirements by following the light rays from the eye point through each pixel of the image plane to the source of the light. This way only light rays that contribute to the image are considered. The computational complexity of the algorithm depends on the resolution of the output image and the scene. The `raytrace` benchmark

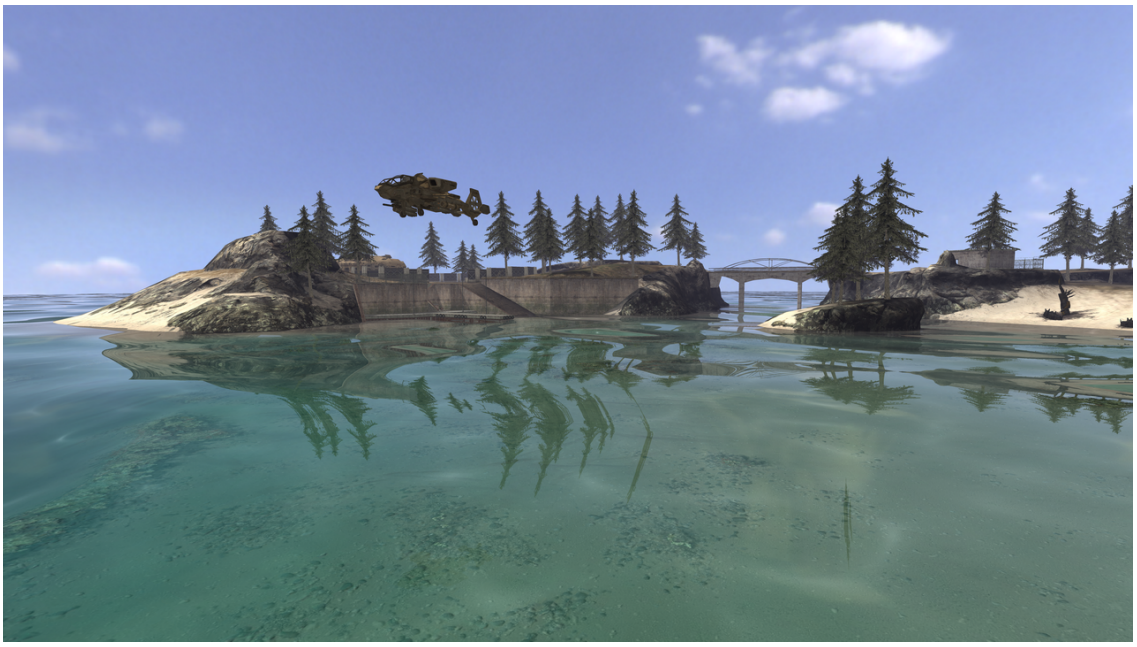

Figure 2.6: Demonstration of the `raytrace` benchmark. The figure shows a screenshot of a raytraced version of *Quake Wars*. The game was modified by Intel to use the ray tracing code included in PARSEC for its rendering. The physically accurate reflections of the scenery on the water surface are clearly visible.

program uses a variety of the ray tracing method that would typically be employed for real-time animations such as computer games because it is optimized for speed rather than realism. The `raytrace` benchmark was included in PARSEC because of the continuing trend towards more realistic graphics in video games and other forms of real-time animation. As of 2009 all major graphics card vendors have announced plans to incorporate ray tracing into their products in one form or another. Commercial computer games adapted to employ ray tracing instead of rasterization have already been demonstrated. A screenshot of a technical demonstration of the `raytrace` benchmark can be seen in Figure 2.6.

All rendering methods try to solve the rendering equation [44], which uses the physical law of conservation of energy to describe the total amount of outgoing light L_o at location x, direction ω and time t with wavelength λ:

$$L_o(x,\omega,\lambda,t) = L_e(x,\omega,\lambda,t) + \int_\Omega f_r(x,\omega',\omega,\lambda,t) L_i(x,\omega',\lambda,t)(\omega' \cdot n)\, d\omega'$$

The total amount of outgoing light L_o is the sum of the emitted light L_e and an integral over all inward directions ω' of a hemisphere that gives the amount of reflected light. f_r is the bidirectional reflectance distribution function which describes the proportion of the incoming light L_i that is reflected from ω' to ω at position x and time t with wavelength λ. The term $\omega' \cdot n$ is the attenuation of inward light. Solving the rendering equation gives theoretically perfect results because all possible flows of light are included,[1] but because of the high computational demand it is only approximated in practice. The ray tracing method does so by sampling the object surfaces at discrete locations and angles as given by the scatter model.

The scatter model describes what happens when a ray hits a surface. In that case the ray tracing method can generate up to three new types of rays: Reflection rays, refraction rays and shadow rays. Reflection rays are created if the surface of the object is shiny. A reflected ray continues to traverse the scene in the mirrored direction from the surface. The closest surface it intersects will be visible as a mirror image on the surface of the reflecting object. If the object is transparent a refraction ray is generated. It is similar to a reflection ray with the notable exception that it enters and traverses the material. Shadow rays are the method that is used by the ray tracing algorithm to determine whether an intersection point is visible or not. Every time a ray intersects a surface, shadow rays are cast into the directions of every light source in the scene. If a shadow ray reaches its light source then the intersection point is illuminated by that light. But if the shadow ray is blocked by an opaque object then the intersection point must be located in its shadow with respect to that light, resulting in a lower light intensity.

To find intersection points quickly ray tracers store the scene graph in a Bounding Volume Hierarchy (BVH). A BVH is a tree in which each node represents a bounding volume. The bounding volume of a leaf node corresponds to a single object in the scene which is fully contained in the volume. Bounding volumes of intermediate nodes fully contain all the volumes of their children, up to the volume of the root node which contains the entire scene. If the bounding volumes are tight and partition the scene with little overlap then a ray tracer searching for an intersection point can eliminate large parts of the scene rapidly by recursively descending in the BVH while performing intersection tests until the correct surface has been found.

[1] The rendering equation does not consider certain physical effects such as phosphorescence, fluorescence or subsurface scattering.

Figure 2.7: The `native` input for the `raytrace` benchmark is a 3D model of a Thai statue with 10 million polygons, which is about the amount of triangles that need to be rendered per frame for modern video games.

The entry point for the rendering algorithm of the `raytrace` benchmark is the `renderFrame` method of the `Context` class. In the parallel case this function merely unblocks all threads, which start executing the `task` method of the `Context` class for each work unit. Work units correspond to tiles on the screen. The work is distributed using the task queue in the `MultiThreadedTaskQueue` class so that the program is dynamically load balanced. The BVH containing the scene is stored in the `m_bvh` object, which is an instance of the `BVH` class. It uses arrays to store the BVH nodes in a compact way so they can be traversed quickly.

For each frame the program starts traversing this scene graph with the `renderTile_With_StandardMesh` method. The method creates the initial rays and then calls `TraverseBVH_with_StandardMesh` to handle the actual BVH traversal. This function is the hot spot of the `raytrace` workload. It is a recursive function by nature, but to eliminate the recursive function calls a user-level stack of BVH nodes is used. The stack is implemented as an array and accessed with the `sptr` pointer. To further optimize the intersection tests the function considers the origins and directions of rays and handles the different cases with specialized code.

Figure 2.7 shows the rendered `native` input. The inputs for the `raytrace` workload are defined as follows:

- `test`: 1×1 pixels, 8 polygons (octahedron), 1 frame
- `simdev`: 16×16 pixels, 68,941 polygons (Stanford bunny), 3 frames
- `simsmall`: 480×270 pixels ($\frac{1}{4}$ HDTV resolution), 1 million polygons (Buddha statue), 3 frames
- `simmedium`: 960×540 pixels ($\frac{1}{2}$ HDTV resolution), 1 million polygons (Buddha statue), 3 frames
- `simlarge`: $1,920 \times 1,080$ pixels (HDTV resolution), 1 million polygons (Buddha statue), 3 frames
- `native`: $1,920 \times 1,080$ pixels (HDTV resolution), 10 million polygons (Thai statue), 200 frames

`Raytrace` can be configured to display its output in real-time.

2.4.10 Streamcluster

This kernel solves the online clustering problem [70]: For a stream of input points, it finds a predetermined number of medians so that each point is assigned to its nearest center. The quality of the clustering is measured by the sum of squared distances (SSQ) metric. Stream clustering is a common operation where large amounts or continuously produced data has to be organized under real-time conditions, for example network intrusion detection, pattern recognition and data mining. The program spends most of its time evaluating the gain of opening a new center. This operation uses a parallelization scheme which employs static partitioning of data points. The program is memory bound for low-dimensional data and becomes increasingly computationally intensive as the dimensionality increases. Due to its online character the working set size of the algorithm can be chosen independently from the input data. `Streamcluster` was included in the PARSEC benchmark suite because of the importance of data mining algorithms and the prevalence of problems with streaming characteristics.

The parallel gain computation is implemented in function `pgain`. Given a preliminary solution, the function computes how much cost can be saved by opening a new center. For every new point, it weighs the cost of making it a new center and reassigning some of the existing points to it against the savings caused by minimizing the distance

$$d(x,y) = |x-y|^2$$

between two points x and y for all points. The distance computation is implemented in function `dist`. If the heuristic determines that the change would be advantageous the results are committed.

The amount of parallelism and the working set size of a problem are dominated by the block size. The inputs of `streamcluster` are defined as follows:

- `test`: 10 input points, block size 10 points, 1 point dimension, 2–5 centers, up to 5 intermediate centers allowed
- `simdev`: 16 input points, block size 16 points, 3 point dimensions, 3–10 centers, up to 10 intermediate centers allowed
- `simsmall`: 4,096 input points, block size 4,096 points, 32 point dimensions, 10–20 centers, up to 1,000 intermediate centers allowed
- `simmedium`: 8,192 input points, block size 8,192 points, 64 point dimensions, 10–20 centers, up to 1,000 intermediate centers allowed
- `simlarge`: 16,384 input points, block size 16,384 points, 128 point dimensions, 10–20 centers, up to 1,000 intermediate centers allowed
- `native`: 1,000,000 input points, block size 200,000 points, 128 point dimensions, 10–20 centers, up to 5,000 intermediate centers allowed

The benchmark writes the computed results to a user-determined output file. By default the file is named `output.txt`.

2.4.11 Swaptions

The `swaptions` application is an Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. The HJM framework describes how interest rates evolve for risk management and asset liability management [34] for a class of models. Its central insight is that there is an explicit relationship between the drift and volatility parameters of the forward-rate dynamics in a no-arbitrage market. Because HJM models are non-Markovian the analytic approach of solving the PDE to price a derivative cannot be used. `Swaptions` therefore employs Monte Carlo (MC) simulation to compute the prices. The workload was included in the benchmark suite because of the significance of PDEs and the wide use of Monte Carlo simulation.

The program stores the portfolio in the `swaptions` array. Each entry corresponds to one derivative. `Swaptions` partitions the array into a number of blocks equal to the number of threads and assigns one block to every thread. Each thread iterates through all swaptions in the work unit it was assigned and calls the function `HJM_Swaption_Blocking` for every entry in order to compute the price. This function invokes `HJM_SimPath_Forward_Blocking` to generate a random HJM path for each MC run. Based on the generated path the value of the swaption is computed.

The following inputs are provided for `swaptions`:

- `test`: 1 swaption, 5 simulations
- `simdev`: 3 swaptions, 50 simulations
- `simsmall`: 16 swaptions, 5,000 simulations
- `simmedium`: 32 swaptions, 10,000 simulations
- `simlarge`: 64 swaptions, 20,000 simulations
- `native`: 128 swaptions, 1,000,000 simulations

`Swaptions` prints the resulting swaption prices to the console.

Figure 2.8: The `native` input for `vips` is a satellite image of the Orion Nebula that was taken by the Hubble Space Telescope.

2.4.12 Vips

This application is based on the VASARI Image Processing System (VIPS) [62] which was originally developed through several projects funded by European Union (EU) grants. The benchmark version is derived from a print on demand service that is offered at the National Gallery of London, which is also the current maintainer of the system. The benchmark includes fundamental image operations such as an affine transformation and a convolution. VIPS was included as a benchmark for two reasons: First, image transformations such as the ones performed by the VASARI system are a common task on desktop computers and should be included in a diverse benchmark suite. Second, VIPS is able to construct multithreaded image processing pipelines transparently on the fly. Future libraries might use concepts such as the ones employed by the VASARI system to make multithreaded functionality available to the user.

The image transformation pipeline of the `vips` benchmark has 18 stages. It is implemented in the VIPS operation `im_benchmark`. The stages can be grouped into the following kernels:

Crop The first step of the pipeline is to remove 100 pixels from all edges with VIPS operation `im_extract_area`.

Shrink Next, `vips` shrinks the image by 10%. This affine transformation is implemented as the matrix operation

$$f(\vec{x}) = \begin{bmatrix} 0.9 & 0 \\ 0 & 0.9 \end{bmatrix} \vec{x} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

in VIPS operation `im_affine`. The transformation uses bilinear interpolation to compute the output values.

Adjust white point and shadows To improve the perceived visual quality of the image under the expected target conditions, `vips` brightens the image, adjusts the white point and pulls the shadows down. These operations require several linear transformations and a matrix multiplication, which are implemented in `im_lintra`, `im_lintra_vec` and `im_recomb`.

Sharpen The last step slightly exaggerates the edges of the output image in order to compensate for the blurring caused by printing and to give the image a better overall appearance. This convolution employs a Gaussian blur filter with mask radius 11 and a subtraction in order to isolate the high-frequency signal component of the image. The intermediate result is transformed via a look-up table shaped as

$$f(x) = \begin{cases} 0.5x & |x| \leq 2.5 \\ 1.5x + 2.5 & x < -2.5 \\ 1.5x - 2.5 & x > 2.5 \end{cases}$$

and added back to the original image to obtain the sharpened image. Sharpening is implemented in VIPS operation `im_sharpen`.

The VASARI Image Processing System fuses all image operations to construct an image transformation pipeline that can operate on subsets of an image. VIPS can automatically replicate the image transformation pipeline in order to process multiple image regions concurrently. This happens transparently for the user of the library. Actual image

processing and any I/O is deferred as long as possible. Intermediate results are represented in an abstract way by partial image descriptors. Each VIPS operation can specify a demand hint which is evaluated to determine the work unit size of the combined pipeline. VIPS uses memory-mapped I/O to load parts of an input image on demand. After the requested part of a file has been loaded, all image operations are applied to the image region before the output region is written back to disk.

A VIPS operation is composed of the main function which provides the public interface employed by the users, the generate function which implements the actual image operation, as well as a start and a stop function. The main functions register the operation with the VIPS evaluation system. Start functions are called by the runtime system to perform any per-thread initialization. They produce a sequence value which is passed to all generate functions and the stop function. Stop functions handle the shutdown at the end of the evaluation phase and destroy the sequence value. The VIPS system guarantees the mutually exclusive execution of start and stop functions, which can thus be used to communicate between threads during the pipeline initialization or shutdown phase. The generate functions transform the image and correspond to the pipeline stages.

The image used as `native` input for `vips` is shown in Figure 2.8. The full list of all sizes of its input images is:

- `test`: 256×288 pixels
- `simdev`: 256×288 pixels
- `simsmall`: $1,600 \times 1,200$ pixels
- `simmedium`: $2,336 \times 2,336$ pixels
- `simlarge`: $2,662 \times 5,500$ pixels
- `native`: $18,000 \times 18,000$ pixels

The benchmark writes the output image to a user-determined file. By default PARSEC users the file name `output.v` for the output data.

2.4.13 X264

The `x264` application is an H.264/AVC (Advanced Video Coding) video encoder. In the 4th annual video codec comparison [92] it was ranked 2nd best codec for its high encoding quality. It is based on the ITU-T H.264 standard which was completed in May 2003 and which is now also part of ISO/IEC MPEG-4. In that context the standard is also known as MPEG-4 Part 10. H.264 describes the lossy compression of a video stream [95]. It improves over previous video encoding standards with new features such as increased sample bit depth precision, higher-resolution color information, variable block-size motion compensation (VBSMC) or context-adaptive binary arithmetic coding (CABAC). These advancements allow H.264 encoders to achieve a higher output quality with a lower bit-rate at the expense of a significantly increased encoding and decoding time. The flexibility of H.264 allows its use in a wide range of contexts with different requirements, from video conferencing solutions to high-definition (HD) movie distribution. Next-generation HD DVD or Blu-ray video players already require H.264/AVC encoding. The flexibility and wide range of application of the H.264 standard and its ubiquity in next-generation video systems are the reasons for the inclusion of `x264` in the PARSEC benchmark suite.

H.264 encoders and decoders operate on macroblocks of pixels which have the fixed size of 16×16 pixels. Various techniques are used to detect and eliminate data redundancy. The most important one is motion compensation. It is employed to exploit temporal redundancy between successive frames. Motion compensation is usually the most expensive operation that has to be executed to encode a frame. It has a very high impact on the final compression ratio. The compressed output frames can be encoded in one of three possible ways:

I-Frame An I-Frame includes the entire image and does not depend on other frames. All its macroblocks are encoded using intra prediction. In intra mode, a prediction block is formed using previously encoded blocks. This prediction block is subtracted from the current block prior to encoding.

P-Frame These frames include only the changed parts of an image from the previous I- or P-frame. A P-Frame is encoded with intra prediction and inter prediction with at most one motion-compensated prediction signal per prediction block. The prediction model is formed by shifting samples from previously encoded frames to compensate for motion such as camera pans.

Figure 2.9: The input frames for `x264` were taken from the animated short film *Elephants Dream*.

B-Frame B-Frames are constructed using data from the previous and next I- or P-Frame. They are encoded like a P-frame but using inter prediction with two motion-compensated prediction signals. B-Frames can be compressed much more than other frame types.

The enhanced inter and intra prediction techniques of H.264 are the main factors for its improved coding efficiency. The prediction schemes can operate on block of varying size and shapes which can be as small as 4×4 pixels.

The parallel algorithm of `x264` uses the pipeline model with one stage per input video frame. This results in a virtual pipeline with as many stages as there are input frames. `X264` processes a number of pipeline stages equal to the number of encoder threads in parallel, resulting in a sliding window which moves from the beginning of the pipeline to its end. For P- and B-Frames the encoder requires the image data and motion vectors from the relevant region of the reference frames in order to encode the current frame, and so each stage makes this information available as it is calculated during the encoding process. Fast upward movements can thus cause delays which can limit the achievable speedup of `x264` in practice. In order to compensate for this effect, the parallelization

model requires that `x264` is executed with a number of threads greater than the number of cores to achieve maximum performance.

`X264` calls function `x264_encoder_encode` to encode another frame. `x264_encoder_encode` uses function `x264_slicetype_decide` to determine as which type the frame will be encoded and calls all necessary functions to produce the correct output. It also manages the threading functionality of `x264`. Threads use the functions `x264_frame_cond_broadcast` and `x264_frame_cond_wait` to inform each other of the encoding progress and to make sure that no data is accessed while it is not yet available.

The videos used for the inputs have been derived from the uncompressed version of the short film *Elephants Dream* [27], which is shown in Figure 2.9. The number of frames determines the amount of parallelism. The exact characteristics of the inputs are:

- `test`: 32×18 pixels, 1 frame
- `simdev`: 64×36 pixels, 3 frames
- `simsmall`: 640×360 pixels ($\frac{1}{3}$ HDTV resolution), 8 frames
- `simmedium`: 640×360 pixels ($\frac{1}{3}$ HDTV resolution), 32 frames
- `simlarge`: 640×360 pixels ($\frac{1}{3}$ HDTV resolution), 128 frames
- `native`: $1,920 \times 1,080$ pixels (HDTV resolution), 512 frames

The encoder writes the output video to a user-determined file. By default PARSEC uses the file name `eledream.264` for all output videos.

2.5 Support for Research

A major reason for the rapid adoption of PARSEC is its support for research and ease of use. Countless examples demonstrate that software will struggle to develop a notable user base if it is hard to use. Lowering the barrier of entry and simplifying its usage has therefore been a design objective for the PARSEC suite right from the beginning.

The goal of PARSEC is to enable research. This requires that PARSEC allows for a high degree of flexibility because the requirements of research projects are usually too heterogeneous to be satisfied by a single benchmarking methodology. The suite is

therefore designed to function like a toolbox which scientists can use to develop their own customized research infrastructure. For example, PARSEC users can choose from a range of threading models, program features or inputs to build and run a benchmark. This is a major difference from industry benchmark suites such as SPEC, which are scoring systems that strive for a high degree of standardization of benchmark compilation and execution to maximize the comparability of performance results across many different architectures.

PARSEC achieves ease of use and flexibility with the following principles:

Automatization Part of the PARSEC distribution are scripts which allow the user to perform standard tasks in an automated and centralized way by using a single, common interface. Researchers can use these scripts for example to build or run benchmarks without having to worry about details. The high degree of automation alleviates benchmark users from tedious tasks and reduces the chance for human errors in repetitive experiments.

Modularity The benchmark suite uses modularity to simplify its handling. New benchmark programs or inputs can be easily added. The scripts of the distribution can automatically detect and use extensions. Binaries of benchmarks are installed in directories that encode important properties such as the used ISA or operating system API so that multiple binary versions of the same workload can be stored and used on demand.

Abstraction PARSEC abstracts from details of the benchmark programs wherever possible. This allows users of the suite to conduct experiments without having to know much more other than that each workload is something that can be compiled and executed. For example, benchmarks can be run via the same generalized command line interface by specifying as little information as the name of the workloads, the binary version, the number of threads as well as the name of the input set (which is again an abstract representation of the actual input files and arguments that will be used).

Encapsulation The details of the inner workings of a workload are encapsulated in standardized configuration files to make the information usable in an easily understandable way. This approach hides the details of how to build or run benchmark pro-

grams and alleviates PARSEC users from having to memorize the peculiarities of individual programs.

Logging PARSEC allows users to recreate their steps by automatically logging important information. The output of benchmark builds and runs is stored in special output files. Likewise, each benchmark binary is automatically amended with a file that contains the exact path and version of the compiler, linker and other build tools that were used for the build. Relevant information such as the optimization level used to generate the individual object files of a program have thus a much higher chance to be retrieved should it become necessary at some point.

These design concepts of the PARSEC benchmark suite simplify its use in practice and make its features more accessible to new users.

2.5.1 PARSEC Framework

The PARSEC benchmark suite can be separated into two major parts: The packages which contain the source code and inputs for each workload and the PARSEC framework, which is everything else. The framework provides the users with the tools, configuration files, documentation and other features that ease the use of the suite in practice. Its purpose is to serve as an interface for the user and provide the glue between packages to ascertain benchmark programs can be built and run smoothly. For example, if the tools of the framework are used to build a benchmark, dependencies between packages such as any library requirements are automatically taken into consideration and resolved without the user having to know the details of the dependency structure.

All the information needed by the tools of the framework to be able to their job is stored in configuration files. The following types of configuration files are used:

PARSEC main configuration Fundamental global properties of the PARSEC suite such as aliases that represent groups of packages are stored in the main configuration file.

System configurations The framework stores all platform-dependent properties in system configurations. Details such as the path to binaries needed by the framework tools or the exact arguments to pass to them are stored in these configuration files. The PARSEC framework can be ported to new platforms in a straightforward way

Hook Function	Description
__parsec_bench_begin	Beginning of benchmark execution
__parsec_roi_begin	Beginning of the Region-of-Interest
__parsec_roi_end	End of the Region-of-Interest
__parsec_bench_end	End of benchmark execution

Table 2.6: The PARSEC hook functions and their meaning.

by defining the required commands in a new system configuration file for that operating system.

Build configurations The details which determine how to build a workload are stored in build configuration files. This includes necessary information such as which compilers to use, the optimization flags to use and which features of the workloads to enable. Benchmark users can make new versions of the benchmark binaries accessible for the framework by creating new build configurations.

Run configurations These configuration files determine how exactly a benchmark program is to be invoked. This includes information about any required input files that are to be extracted and put into the working directory of the benchmark as well as the command line arguments to use. Each standardized input set of PARSEC has exactly one run configuration file for each workload.

All configuration files use a standardized human-readable format that allows benchmark users to customize the suite for their needs.

2.5.2 PARSEC Hooks

The PARSEC Hooks API is an instrumentation API that has been defined to allow the rapid insertion of instrumentation and analysis code into the benchmark programs. All PARSEC workloads support the instrumentation API and can be built so they call special hook functions at predefined locations. PARSEC users can easily insert their own code at those locations by writing a library that implements these functions or by modifying the default hooks library offered by the PARSEC suite.

The hook functions currently supported are given in Table 2.6. A default implementation is provided which implements frequently used functionality such as time measurement. The hook functions defining the Region-of-Interest have special significance

because they are needed to obtain measurements with the simulation inputs that have not been skewed. As was explained in Chapter 5 the scaling of the simulation inputs caused an inflation of the size of the initialization and shutdown phase that is not representative of real program behavior. This skew can be accounted for by only measuring the Region-of-Interest.

2.6 Conclusions

The PARSEC benchmark suite is designed to provide parallel programs for the study for CMPs. This chapter described the composition of the suite and its workloads. The suite satisfies the five requirements a modern benchmark suite for multiprocessors should have: All workloads are multithreaded so that they can take advantage of the increased performance potential of next-generation CMPs. The suite focuses on emerging desktop and server applications that take into account current trends in computing. PARSEC includes a diverse selection of programs from different application domains that implement a range of threading and parallelization models. It is one of the first suites to include the pipeline model. PARSEC workloads use state-of-art algorithms and the suite supports research. This combination of criteria make the PARSEC benchmark suite a suitable choice for computer architecture research.

Chapter 3

Comparison of PARSEC with SPLASH-2

3.1 Introduction

The Princeton Application Repository for Shared-Memory Computers (PARSEC) is a collection of multithreaded benchmarks which extends the spectrum of parallel workloads researchers can choose from. It is the outcome of a joint venture between Intel and Princeton University that seeks to provide the research community with an up-to-date collection of modern workloads for studies of chip multiprocessors (CMPs). The new benchmark suite immediately aroused the interest of researchers around the world who have started to use it for their work. Hundreds of papers with results obtained through PARSEC have already been published. This motivates a fundamental question:

What distinguishes PARSEC from other benchmark suites?

Several multithreaded benchmark suites are available. SPLASH-2 [96] and SPEC OMP2001 include workloads from different domains but focus on High-Performance Computing. BioParallel [41] is composed of bioinformatics programs. ALPBench [55] is a suite of multimedia workloads. MineBench [65] was created to study data mining. With PARSEC, researcher now have a new option and need to understand how the selection of this benchmark suite can impact their results. Other scientists might face the challenge

to interpret PARSEC results and seek ways to apply their existing knowledge of other workloads to the new benchmarks.

To help other researchers understand PARSEC this chapter compares the suite to the SPLASH-2 selection of programs. SPLASH-2 is probably the most commonly used suite for scientific studies of parallel machines with shared memory. According to Google Scholar, its characterization study [96] was cited more than 1400 times. Like PARSEC it is one of few parallel suites that are not limited to a single application domain. Its wide use and the thorough understanding researchers have of these workloads make it an excellent candidate for a comparison. PARSEC tries to provide a wider selection of workloads than SPLASH-2.

This chapter makes four contributions:

- Compares PARSEC with SPLASH-2 to determine how much the program selections of the two suites overlap. Significant differences exist that justify an overhaul of the popular SPLASH-2 benchmark suite.
- Identifies workloads in both suites that resemble each other that can help researcher to interpret results. A few benchmarks of the two suites have similar characteristics.
- Demonstrates how current technology trends are changing programs. The direct comparison of the PARSEC suite with SPLASH-2 shows that the proliferation of CMPs and the massive growth of data have a measurable impact on workload behavior.
- Shows that workloads using the pipeline programming model have different characteristics from other programs, which justifies their inclusion in a mix of benchmark programs.

The scope of this chapter is the parallel aspects of the behavior of multithreaded workloads on CMPs. Moreover, single-core characteristics are not considered because this is not the intended use of either benchmark suite. The focus of this study is on redundancy within and between the suites.

The work presented in this chapter was previously published in [9, 13].

Program	Application Domain	Problem Size
barnes	High-Performance Computing	65,536 particles
cholesky	High-Performance Computing	tk29.O
fft	Signal Processing	4,194,304 data points
fmm	High-Performance Computing	65,536 particles
lu	High-Performance Computing	1024×1024 matrix, 64×64 blocks
ocean	High-Performance Computing	514×514 grid
radiosity	Graphics	large room
radix	General	8,388,608 integers
raytrace	Graphics	car
volrend	Graphics	head
water	High-Performance Computing	4096 molecules

Table 3.1: Overview of SPLASH-2 workloads and the used inputs.

3.2 Overview

The SPLASH-2 suite is one of the most widely used collections of multithreaded workloads [96]. It is composed of eleven workloads, three of which come in two implementations that feature different optimizations. I provide an overview in Table 3.1. When SPLASH-2 was released at the beginning of the 90s, parallel machines were still a relatively uncommon and expensive type of computers. Most of them were owned by well funded government and research institutions where they were primarily used to work on scientific problems. The composition of the SPLASH-2 suite reflects that. The majority of workloads belong to the High-Performance Computing domain.

The workload composition of the PARSEC suite differs significantly from SPLASH-2. Since the release of SPLASH-2 parallel computing has reached the mainstream. The wide availability of CMPs has turned multiprocessor machines from an expensive niche product into a commodity that is used for problems from an increasingly wide range of application domains. This fact has influenced the PARSEC program selection. The suite includes benchmarks from many different areas such as enterprise servers, data mining and animation.

Unlike SPLASH-2, PARSEC already includes input sets that reflect computing problem sizes suitable for current microarchitecture studies. The input set given as reference for SPLASH-2, however, cannot be considered adequate for simulations anymore due to its small size and higher age. Where possible a combination of profiling and timing on

real machines was used to determine inputs for SPLASH-2 that have computational demands similar to the PARSEC inputs. In order to preserve the comparability of different programs the same input was used for workloads that solve the same problem, even if the computational requirements would have allowed the selection of a bigger problem size. In each case the two versions of `lu`, `ocean` and `water`, but also `barnes` and `fmm` have therefore the same input size. Table 3.1 shows the inputs that were chosen for this study.

3.3 Methodology

The following methodology is used to gather and analyze data about the behavior of the workloads: First, a set of interesting characteristics is identified. Execution-driven simulation is then used to obtain the data relevant for the characteristics. Finally, standard statistical methods were applied to the data to compute the similarity of the workloads.

3.3.1 Program Characteristics

Both PARSEC and SPLASH-2 are aimed at the study of parallel machines. A comprehensive benchmark suite for single processor systems already exists with SPEC CPU2006 [83]. The focus of this study is therefore on the parallel behavior of the programs, which means that characteristics were primarily chosen which reflect how threads communicate with each other on a CMP and how data is shared. The selection of interesting program characteristics and how they were measured largely follows the methodology established by previous work on characterization of multithreaded programs [11, 38, 55, 96].

The chosen characteristics are given in Table 3.2. To capture the fundamental program properties a set of four instruction characteristics were included that were normalized to the total number of instructions: The number of floating point operations, ALU instructions, branches and memory accesses. Threads running on a CMP use shared caches to communicate and share data with each other. Another five characteristics were thus chosen that reflect properties related to data usage and communication such as the total working set size or how intensely the program works with the shared data. These characteristics are the data cache miss rate, what percentage of all cache lines is shared for reading and what percentage for writing, the ratio of memory references that reads from shared cache lines and the ratio that writes to them.

Characteristic	Type
Floating point operations per instruction	Instruction
ALU operations per instruction	Instruction
Branches per instruction	Instruction
Memory references per instruction	Instruction
Cache misses per memory reference	Working Set
Fraction of cache lines shared	Sharing
Fraction of cache lines shared and written to	Sharing
Accesses to shared lines per memory reference	Sharing
Writes to shared lines per memory reference	Sharing

Table 3.2: Characteristics chosen for the redundancy analysis. Instruction metrics are based on totals across all cores for the whole program.

One difficulty in extracting properties related to cache usage is that the behavior of the program might change with the cache size. For example, shared data might get displaced by more frequently used private data [11] if the cache is too small to contain the whole working set. It is therefore necessary to collect data for a sufficiently large range of cache sizes. In order to avoid that unrealistic architecture parameters skew the data towards aspects of the program behavior not relevant for future CMPs, the experiments were limited to 8 cache sizes ranging from 1 MB to 128 MB. This approach results in the following 44 characteristics:

Instruction Mix 4 characteristics that describe which instructions were executed by the program

Working Sets 8 characteristics providing information about working set sizes

Sharing 32 characteristics describing how much of the working set is shared and how intensely it is used

3.3.2 Experimental Setup

The data was obtained with Pin [58]. Pin is comparable to the ATOM toolkit [86] for Compaq's Tru64 Unix on Alpha processors. It employs dynamic binary instrumenta-

tion to insert routines into the instruction stream of the program under analysis. To obtain information about the impact of different cache sizes, CMP$im [41] was employed. CMP$im is a plug-in for Pin that simulates the cache hierarchy of a CMP.

The numbers for the working set and sharing characteristics were collected by simulating a single shared cache for a CMP. This method was used because I am interested in fundamental program properties, not processor features. This approach abstracts from the architectural details of the memory hierarchy while still capturing the fundamental properties of the program. It is a common method for the analysis of multithreaded programs [11, 38, 55, 96].

An 8-way CMP with a single cache shared by all cores was simulated. The cache was 4-way associative with 64 byte lines. Its capacity was varied from 1 MB to 128 MB to collect the characteristics for different cache sizes. The experiments were conducted on an 8-way SMP with Intel 64-bit CPUs running a `Linux 2.6.9` kernel. All programs were compiled with `gcc 4.2.1`. The compiler chosen was `gcc` because of its wide-spread use. It is usually the compiler of choice for many non-scientific workloads. The entire runtime of all programs was simulated.

3.3.3 Removing Correlated Data

Characteristics of real-world programs might be correlated. For example, the behavior of programs on CMPs with two caches of similar size might be almost identical. Correlated characteristics can skew the redundancy analysis. It is therefore necessary to eliminate correlated information with principal component analysis (PCA) [26]. PCA is a common method used for redundancy analysis [28, 31, 42, 51, 73]. First, the data is mean-centered and normalized to make it comparable. PCA is then employed to remove correlated information and reduce the dimensionality of the data. PCA computes new variables – the principal components (PCs) – that are linear combinations of the original variables. The vectors computed in that manner have decreasing variance, i.e., the amount of information in each vector decreases. In order to decide objectively how much information to keep, Kaiser's Criterion was used to choose how many PCs to eliminate. This approach keeps only the top few PCs that have eigenvalues greater than or equal to one. The resulting data is guaranteed to be uncorrelated while capturing most of the information from the original variables.

The principal components can be visualized with a scatter plot. Each workload defines a point in the PCA space based on its characteristics. This method is only feasible with up to three dimensions, which means that it might not be possible to consider all of the information. The PCs created by the PCA are an alternative representation of the input data with the property that information is concentrated in the first PCs and redundancy in the latter ones. This means that a three-dimensional scatter plot contains the maximum amount of information that can be expressed in three dimensions.

3.3.4 Measuring Similarity

Hierarchical clustering was employed to group similar programs into clusters. The Euclidean distance between the program characteristics is a measure for the similarity of the programs. This approach is a common way to process the output of a PCA [28,31,42,73]. Hierarchical clustering works as follows:

1. Assign each workload to its own cluster.
2. Compute the pair-wise distances of all clusters.
3. Merge the two clusters with the smallest distance.
4. Repeat steps 2 - 3 until only a single cluster is left.

The output of the hierarchical clustering algorithm can be visualized with a dendrogram. The vertical axis lists all workloads, the horizontal axis is the linkage distance. Each joint in the dendrogram corresponds to a merge step of the clustering algorithm. Its projection onto the horizontal axis shows how similar two clusters were when they were merged. Clusters with very dissimilar workloads will be merged late, their joint will be close to the root. Programs with very similar characteristics on the other hand will be merged early. Their joint will be close to the leaves, which represent the individual workloads.

3.3.5 Interpreting Similarity Results

The similarity information is analyzed with the help of dendrograms and scatter plots. Dendrograms are used to provide an objective overview. Interesting findings are analyzed further with scatter plots by focusing on subsets of the data. This section explains how the visualized information can be interpreted.

Dendrograms

The similarity information shown in a dendrogram can be used in two different ways. First, for a selection of benchmark programs it is usually desirable that their characteristics cover a wide range, which means the workloads are very dissimilar and far away from each other in a dendrogram. The dendrogram for a selection of workloads which satisfy this criterion well would branch out close to the root into a wide tree. Each branch would be very narrow because it would be composed of a single workload. This type of dendrogram structure indicates that the analyzed workloads do not resemble each other well.

The second way to use a dendrogram is to identify replacements for workloads. A replacement should be very similar to the benchmarks which it replaces, which means in a dendrogram it will merge close to the leaves which are formed by the individual programs. Workloads frequently form local clusters that merge early. Any program from the cluster is a suitable representative for the entire group, and the closer to the leaves the cluster forms the more similar the workloads are. For a benchmark suite it is usually not desirable that its programs resemble each other much. A dendrogram can be used to optimize a benchmark selection by eliminating similar programs from local clusters until the resulting program selection is very dissimilar.

For a comparison of two benchmark suites such as the one presented in this chapter a dendrogram can provide the following insights:

- The dendrogram indicates how much the covered characteristics space of the two program selections overlap. If parts of the two suites are very similar then pairs of workloads from the two suites will merge early. If the suites cover different areas of the characteristics space then their workloads will merge late.

- The information in a dendrogram can be used to identify workloads from the compared suites which are mutually redundant. These programs will merge early. This information is useful for a new suite because it allows one to explain the behavior of new benchmarks by relating them to benchmarks which are already well-understood. However, a new benchmark suite should generally provide programs which are new and dissimilar to existing benchmarks to justify its existence.

- The data in a dendrogram indicates how diverse the compared suites are in a direct comparison and how diversity is added by the individual workloads. A suite with little diversity will have many local clusters that merge early, a property which is generally not desirable for a benchmark program selection.

Scatter Plots

Another way to visualize the similarity information is by plotting the first three principal components in an unmodified form as a three-dimensional scatter plot. This approach gives insights into the relationship between the workloads at the expense of accuracy. Unlike dendrograms, which only exploit the distances between points, scatter plots preserve the information about the location of the points and make them visible. Scatter plots can be used to identify any structure in the input data. If two different types of workloads occupy different areas of the PCA space, they must be different in a systematic way. However, scatter plots visualize only a subset of the available information and should hence not be used to infer information about workload similarity. Programs appearing close in the plot might in fact be far away if all relevant dimensions are considered. Proximity in a scatter plot does therefore not prove similarity.

3.4 Redundancy Analysis Results

This section employs PCA and hierarchical clustering to analyze how redundant the PARSEC and SPLASH-2 workloads are. The following three questions are answered:

- How much do the two program collections overlap?

- In particular, which workloads of the PARSEC suite resemble which SPLASH-2 codes?

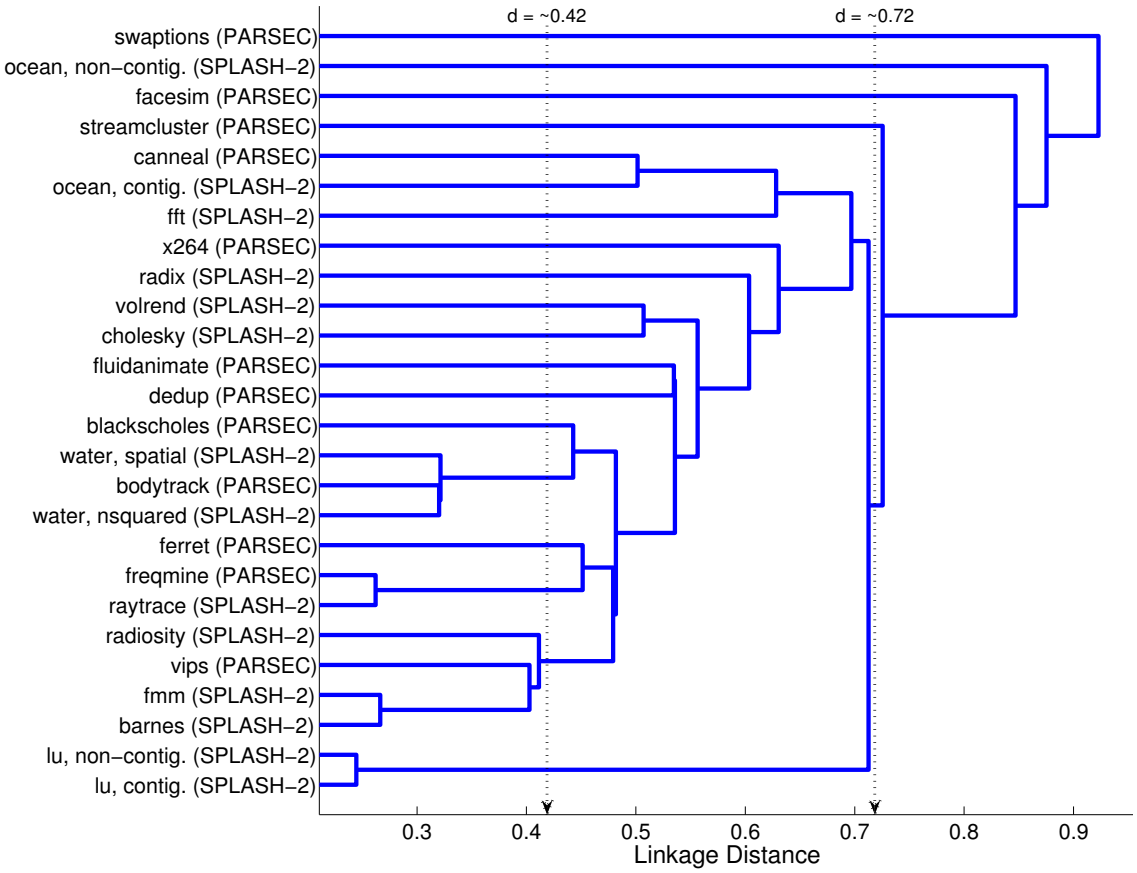

Figure 3.1: Similarity of PARSEC and SPLASH-2 workloads. The two vertical arrows are used for illustration purposes. SPLASH-2 codes tend to cluster early (distance $d <\sim 0.42$), PARSEC includes a larger number of diverse workloads (distance $d >\sim 0.72$).

- Which benchmark suite is more diverse?

Answers to those questions are obtained by analyzing the redundancy within and between the two benchmark suites.

My first step was to analyze both benchmark suites separately to measure their diversity by computing the total variance of their characteristics. It is almost the same for both suites: SPLASH-2 characteristics have a variance of 19.55, for PARSEC the value is 18.98. However, the variance does not take into account how programs add diversity. Moreover, workloads with almost identical characteristics that deviate substantially from the mean will artificially inflate the variance without contributing much beyond the inclusion of only one of these programs. We will see that this is the case with the two `lu` codes. A more detailed analysis is therefore necessary before conclusions can be drawn.

My second step was a direct comparison. To compare the suites directly with each other, all workloads are analyzed jointly using a single PCA. This approach guarantees that the PCA weighs all characteristics equally for all programs, since different data will generally result in different correlation and hence different PCs. Having equal weights for the characteristics of all workloads makes it possible to compare the benchmarks directly. The PCA chose 10 principal components that retain 74.73% of the variance. Figure 3.1 shows the result as a dendrogram containing all PARSEC and SPLASH-2 workloads. PARSEC exhibits substantially more diversity than SPLASH-2. The dendrogram shows that several SPLASH-2 programs form clusters early on. As one would expect, most of the SPLASH-2 programs that come in two versions exhibit significant amounts of redundancy (`lu` and `water`). Only the two `ocean` codes are noticeably different. In fact, the non-contiguous version of `ocean` is the one least similar to any other SPLASH-2 workloads. This is mainly a consequence of intense inter-core communication that is caused by its two-dimensional data layout. With a 4 MB cache, 46% of all memory references of the non-contiguous version of `ocean` go to shared data. About one in three of these references is a write. That is a more than three times higher ratio of shared writes than the next highest one of any SPLASH-2 program. This difference is caused by optimizations for machines with distrubted shared memory and will be discussed in more detail in Section 3.7.1.

Before any PARSEC workloads start to form clusters with each other, the algorithm has already identified 3 groups of workloads containing 7 SPLASH-2 programs in total that exhibit similar characteristics. These clusters have a linkage distance less than $d =\sim 0.42$. As was mentioned earlier workload pairs consisting of two versions of the same program tend to form clusters (`lu` and `water`). Obviously the differences between both versions do not noticeably affect their characteristics. The programs `radiosity`, `fmm` and `barnes` form another cluster. `Vips` is the PARSEC workload most similar to them. These benchmarks tend to use a limited number of branches (no more than 11.24% of all instructions). They have medium-sized working sets and benefit from additional cache capacity up to 16 MB. At that point their miss rates fall below 0.1%. `Bodytrack` is identified as the PARSEC workload most similar to the `water` programs. Programs of that cluster use about the same amount of floating point operations (between 29.88% and 31.97% of all instructions) and memory accesses (between 27.83% and 35.99% of all instructions). About half of all memory references are used to access shared data once

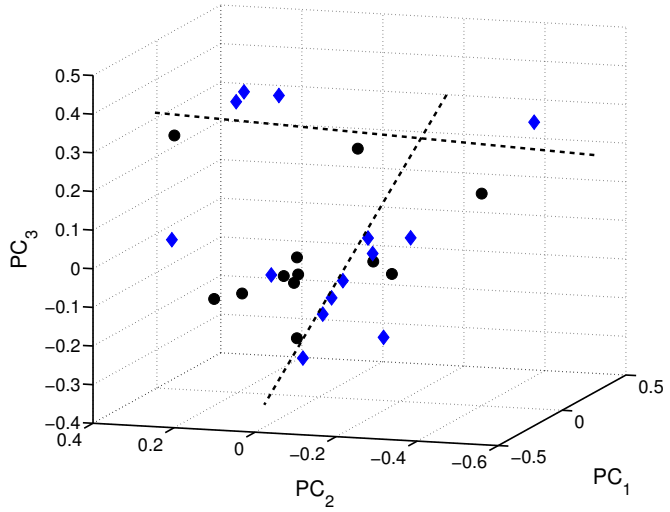

Figure 3.2: Scatter plot of all workloads using the first three PCs of all characteristics. PARSEC (black circles) and SPLASH-2 workloads (blue rhombi) tend to populate different regions of the characteristics space.

the cache capacity is sufficiently large.

The first pair of PARSEC workloads to be assigned to the same cluster are `bodytrack` and `blackscholes`. They exhibit a linkage difference of about 0.45. The algorithm then identifies a large number of workloads with similar distances. By the time cluster pairs with a distance of about 0.72 are considered, most workloads have been assigned to larger clusters. A distance within that range is the common case for both suites. No obvious similarities can be found anymore between programs clustering in that range.

Several workloads exist that are very different from all other programs in both suites (`swaptions`, the non-contiguous version of `ocean`, `facesim` and `streamcluster`). These programs have a high linkage distance of more than 0.72 to any other cluster of programs and can be considered unique within the analyzed program collection. All but one of these workloads are PARSEC programs. If the two `lu` kernels are treated as a single program, they can also be added to this enumeration, however they have a significantly lower distance to the remainder of the suites than, for example, `swaptions` or the non-contiguous version of `ocean`.

Only two PARSEC programs were identified that resemble some of the SPLASH-2 codes (`bodytrack` and `vips`). The similarity within clusters of SPLASH-2 workloads is often greater than the similarity to most other PARSEC workloads. This finding indicates

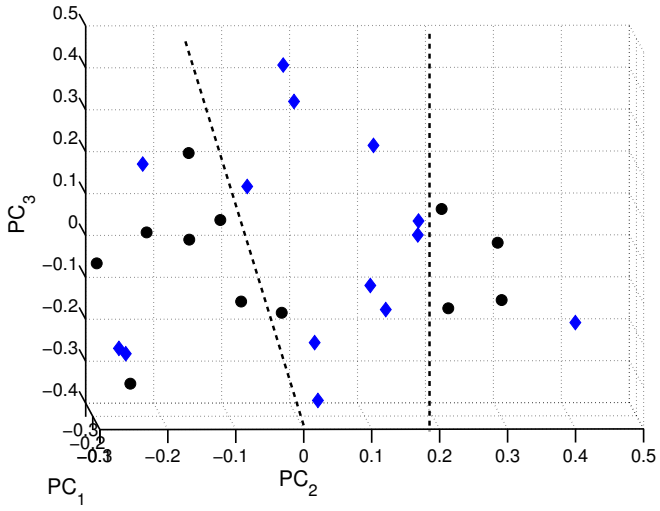

Figure 3.3: Scatter plot using only the instruction mix characteristics. SPLASH-2 workloads (blue rhombi) form a single, major cluster in the first three PC dimensions that contains virtually no PARSEC programs (black circles).

that the two suites cover fundamentally different types of programs. Figure 3.2 is a scatter plot of all workloads using the first three PCs. Lines were added to indicate the regions which are referred to. These lines are not meant to be boundaries and their exact location is not relevant for my conclusions. They are only an aid to visualize tendencies. From Figure 3.2 it can be seen that all but three PARSEC workloads group in the lower left part of the chart, while all but two SPLASH-2 programs are located in the remaining space. Obviously, PARSEC and SPLASH-2 have little overlap.

The analysis shows that on modern CMPs, the PARSEC suite contains significantly more diversity than SPLASH-2. Benchmarks of the SPLASH-2 suite tend to cluster early while the PARSEC suite contains a larger number of unique benchmarks. Moreover, PARSEC and SPLASH-2 workloads are fundamentally different, as shown by the scatter plot.

3.5 Systematic Differences

To identify the reason why the two suites differ, an analysis of subsets of the characteristics was performed. All metrics were broken up into groups reflecting the instruction mix, working sets and sharing behavior of the programs, which were analyzed separately

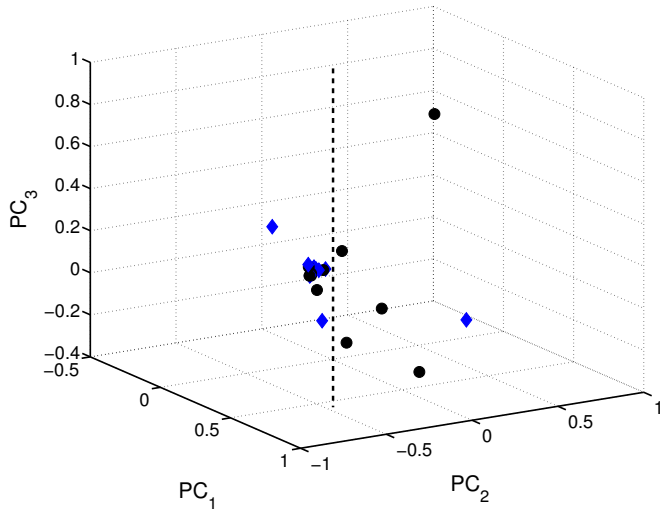

Figure 3.4: Scatter plot using only the working set characteristics. At least half of the PARSEC workloads (black circles) have substantially different working set properties in the first three PC dimensions, whereas only one of the SPLASH-2 programs (blue rhombi) is visibly unique.

from each other. This approach allows one to determine which types of characteristics are the reason for the differences. The results are presented in Figures 3.3 - 3.5. Lines were again added to identify the discussed regions.

The instruction mix of the workloads will be discussed first. It differs significantly between PARSEC and SPLASH-2. Figure 3.3 presents a scatter plot of the first three PCs derived from the four instruction mix characteristics. As can be seen from the figure, SPLASH-2 codes tend to populate the area in the middle. PARSEC programs can primarily be found in the outer regions. The overlap of the suites is small. This result is not surprising considering that the two suites include programs from different domains.

The next aspect which is analyzed is the working sets. About half of all PARSEC workloads have noticeably different working set characteristics from all other programs. A scatter plot based on the eight miss rates can be seen in Figure 3.4. The analysis of the first three PCs shows that there exists a tight cluster in the first three dimensions to which almost all SPLASH-2 codes and many PARSEC workloads belong. Only one SPLASH-2 program is visibly different, but multiple PARSEC workloads have noticeably different working set properties.

The sharing behavior is one of the most important properties of a multithreaded program on a CMP. Similarities between the two suites seem to exist, albeit with differ-

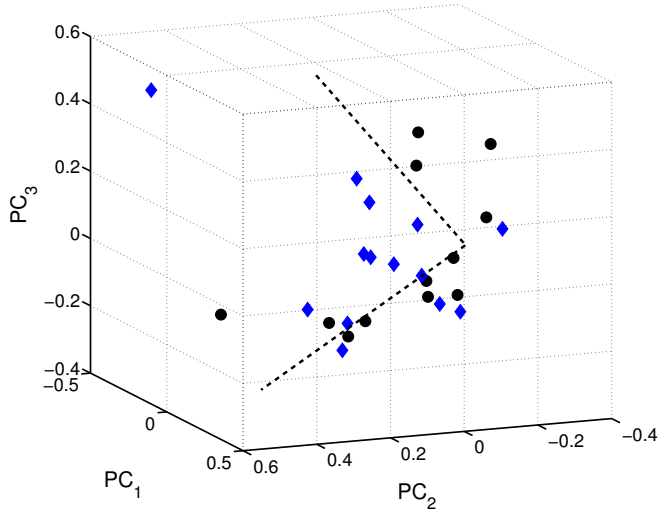

Figure 3.5: Scatter plot using only the sharing characteristics. PARSEC benchmarks (black circles) and SPLASH-2 programs (blue rhombi) tend to populate different areas of the first three PC dimensions of the characteristics space.

ent tendencies. Figure 3.5 shows a scatter plot with only the 32 sharing characteristics. Benchmarks of the SPLASH-2 suite can predominantly be found in the area on the left side of the figure. PARSEC programs tend to populate the areas on the right and bottom half. Some overlap seems to exist in the lower half of the figure around the horizontal line where a sufficient number of workloads of both suites is located to indicate that commonalities might exist. However, these similarities can only exist between approximately half of the programs in each suite.

The analysis shows that there is no single source for the differences of the two suites. The program collections exhibit dissimilarities in all studied characteristics. No single property can be identified that can be considered the main reason for the differences.

3.6 Characteristics of Pipelined Programs

This section discusses how the use of the pipeline programming model has affected the characteristics of the PARSEC workloads. The analysis shows that there are substantial, systematic differences, which suggests that researchers can improve the diversity of their benchmark selection by including pipelined programs.

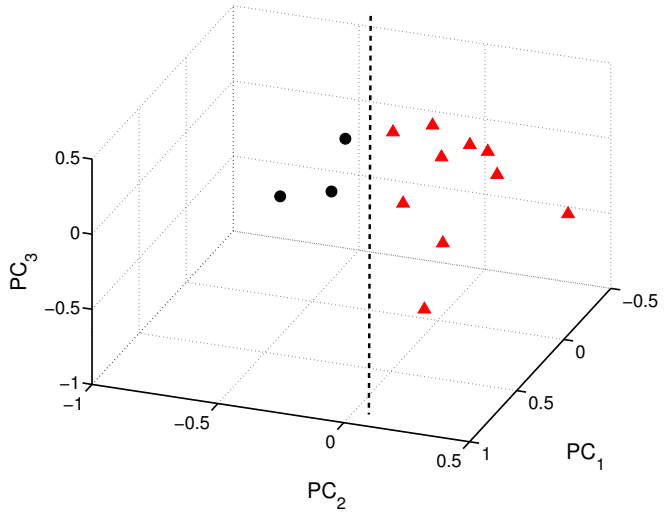

Figure 3.6: Comparison of the first three principal components of all characteristics of the PARSEC workloads. Pipeline workloads are represented by blue dots, all other workloads by red triangles. The data shows significant systematic differences between the two types of programs.

3.6.1 Experimental Setup

To study the differences within the PARSEC suite a more detailed methodology was chosen. The results in this section were obtained with Simics. A total of 73 characteristics were measured for each of the workloads: 25 characteristics describing the breakdown of instruction types relative to the amount of instructions executed by the program, 8 characteristics encoding the working set sizes of the program with cache sizes ranging from 1 MB to 128 MB, and 40 characteristics describing the sharing behavior of the program. The data was statistically processed as described before.

3.6.2 Experimental Results

Figure 3.6 shows the first three principal components derived from all studied characteristics. As can be seen the three workloads which employ the pipelining model (represented by black dots) occupy a different area of the PCA space as the rest of the PARSEC programs (represented by red triangles). The PCA space can be separated so that the different clusters become visible, as is indicated by the dashed line which was added as a visual aid.

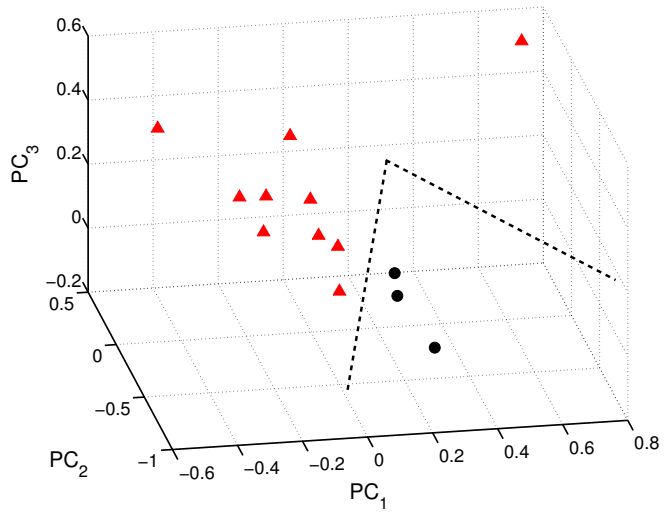

Figure 3.7: Comparison of the first three principal components of the sharing characteristics of the PARSEC workloads. Pipeline workloads are represented by black dots, all other workloads by red triangles. Systematic differences in sharing are a major source for the different behavior of pipelined workloads.

A further investigation of the individual characteristics reveals the sharing behavior of the workloads as a major source for the differences. Figure 3.7 presents a scatter plot that was obtained with just the sharing characteristics. As can be seen the PCA space of the sharing characteristics can also be separated so that the two types of workloads occupy different areas. However, the difference seems to be less pronounced than in the previous case which considered all characteristics.

The remaining characteristics which encode the instruction mix and working sets of the workloads also exhibit a small tendency to group according to the parallelization model of the workloads. However, the differences are much smaller in scope and separation. The aggregate of these differences appears to be the reason for the clearer separation seen in Figure 3.6 compared to Figure 3.7.

The analysis suggests that pipelined programs form their own type of workload with unique characteristics. Their behavior is different enough to warrant their consideration for inclusion in a mix of benchmarks for computer architecture studies.

3.7 Objectives of PARSEC

PARSEC was designed with the objective to capture recent trends in computing. The remarkably small overlap between PARSEC and SPLASH-2 indicates that these developments might have an impact on workloads that fundamentally alters their characteristics. This section analyzes the data from the perspective of two technology trends that have influenced the PARSEC suite: The proliferation of CMPs and the growth of the world's data. PARSEC workloads have been optimized to take advantage of CMPs. Its inputs capture the increasing amount of data that is globally available. SPLASH-2, on the other hand, was created before either of those trends could have an impact on its composition or the design of its programs. The section discusses how both trends are affecting programs. The goal is to provide a basic understanding of the results of the last section.

Figures 3.8 and 3.9 show the miss rates and ratio of shared writes to all accesses of the workloads. This information is used for the analysis of the impact of the trends. Only a fraction of the information considered by the clustering algorithm for the redundancy analysis can be shown here because the amount of data exceeds what can be comprehended by humans. The averages across all cache sizes and details for a selected cache size for both characteristics are shown. Different cache sizes for the detailed breakdowns in the two cases are used because the shared write ratio is positively correlated with the cache size, unlike miss rates, which are negatively correlated. Different cache sizes from opposite ends of the used spectrum reveal more information about the program behaviors.

3.7.1 Chip Multiprocessors

CMPs have become ubiquitous. They integrate multiple processing cores on a single die. The implications of this integration step are twofold: First, it has turned multiprocessors into a widely available commodity that is used to run an increasingly diverse spectrum of programs. The PARSEC suite takes this into account and includes workloads from a wider range of application domains. This is one of the reasons for the increased diversity of the suite. Some characteristics such as the instruction mix seem to be directly affected by that. Second, the trend to CMPs changes the cost model that is employed for program optimizations: On-chip traffic between cores is fast and inexpensive. Off-chip accesses to main memory are costly and usually limited by the available off-chip bandwidth. PARSEC workloads have been adapted to this cost model. SPLASH-2 programs, however,

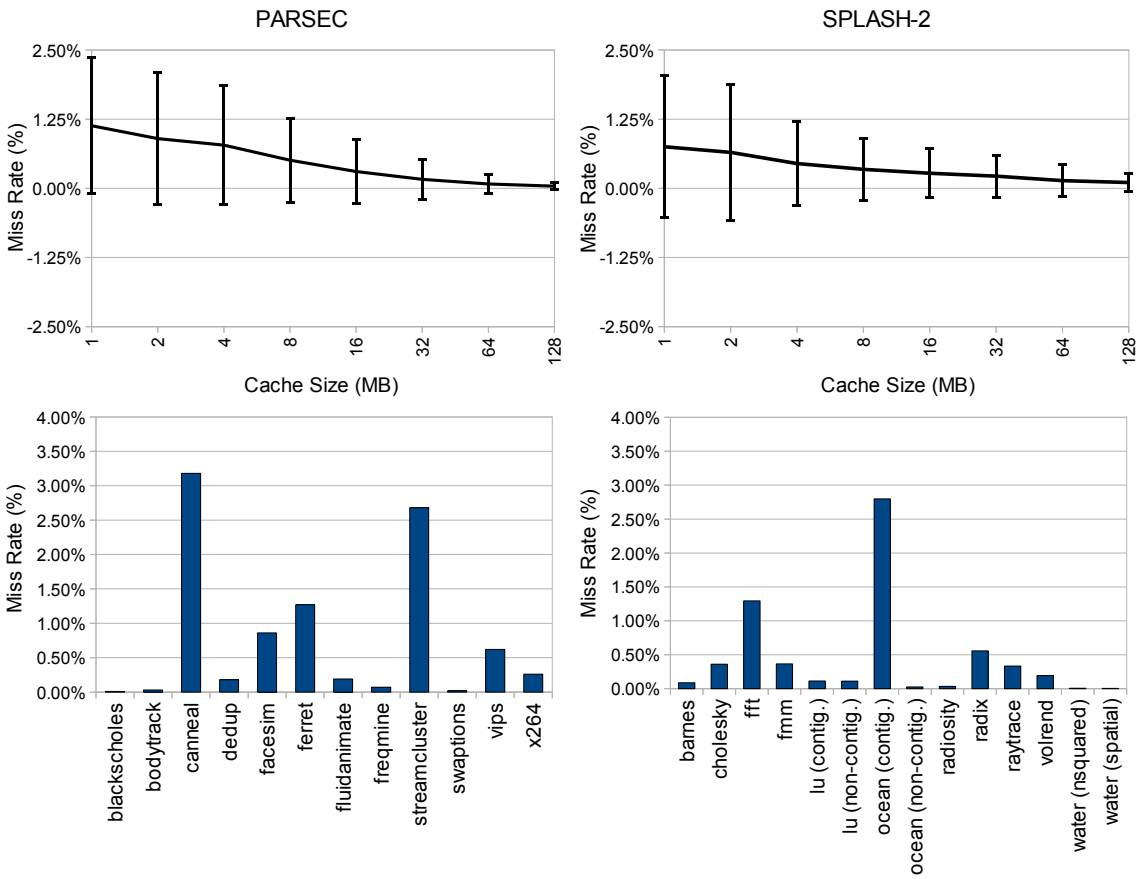

Figure 3.8: Miss rates of PARSEC and SPLASH-2 workloads. The top charts show the averages of the workloads and the standard deviation for all cache sizes. The bottom charts give the detailed miss rates of the programs for a 4 MB cache. Miss rates of PARSEC workloads are noticeably higher for caches up to 16 MB.

have been optimized for systems with distributed shared memory. They assume a large amount of local memory that can be accessed at relatively little cost while communication with other processing nodes requires I/O and is expensive. This is the opposite of the CMP cost model and can have a significant negative impact on the behavior of the program on CMPs as will be demonstrated using the `ocean` codes.

High Impact of Optimizations

`Ocean` is a program that simulates large-scale ocean movements. SPLASH-2 provides two versions that solve the same problem but employ a different memory layout: The non-contiguous implementation manages the grid on which it operates with two-dimensional

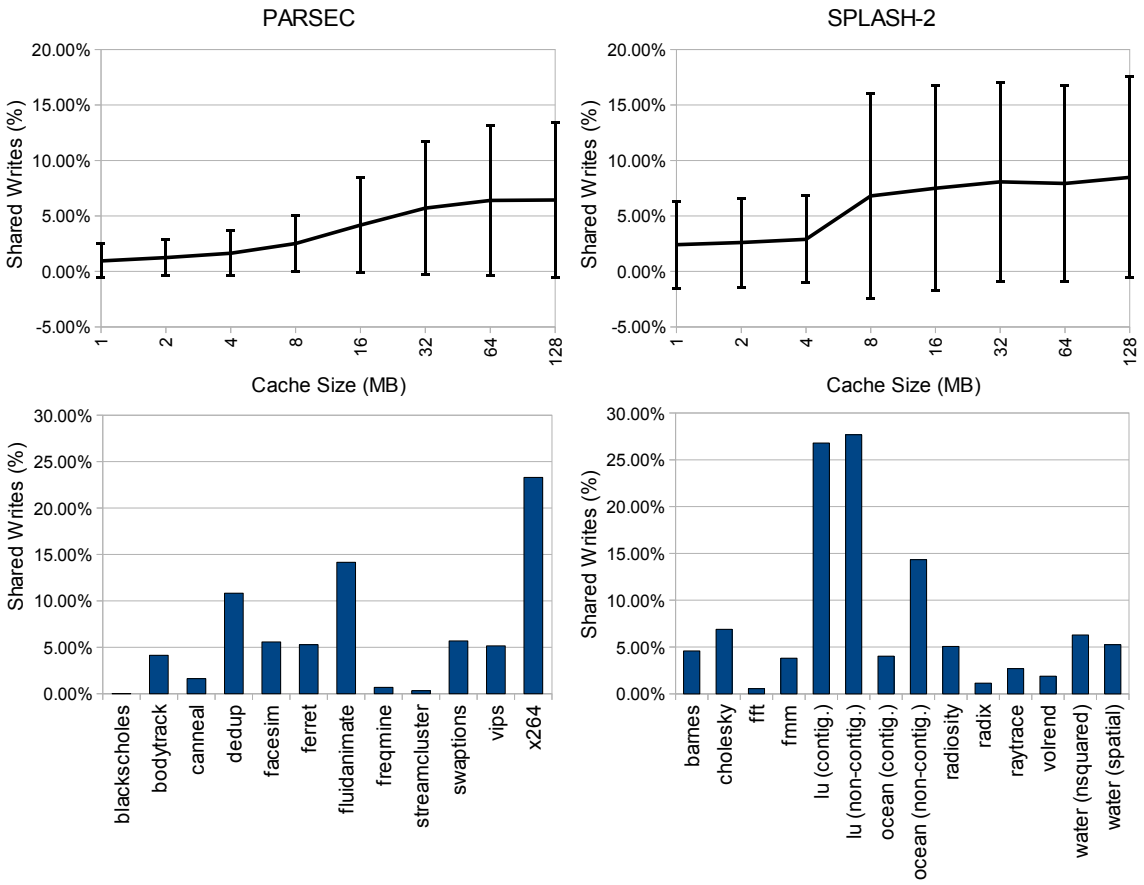

Figure 3.9: Ratio of shared writes to all memory accesses of PARSEC and SPLASH-2. The top charts show the averages of the workloads and the standard deviation for all cache sizes. The bottom charts give the detailed ratios of the programs for a 64 MB cache. The ratio of SPLASH-2 is dominated by the two `lu` workloads.

arrays. This data structure prevents that partitions can be allocated contiguously. The contiguous version of the code implements the grid with three-dimensional arrays. The first dimension specifies the processor which owns the partition so that partitions can be allocated contiguously and entirely in the local memory of machines with distributed shared memory. Figures 3.8 and 3.9 show that this optimization lowers the number of shared writes at the cost of a much higher miss rate on CMPs. The effect is significant. The contiguous version of `ocean` has a shared write ratio that is about 3-5 times lower than the contiguous version for all cache sizes. Its miss rate, however, is about two orders of magnitude higher for small caches. It decreases to 0.13% compared to only 0.03% for the non-contiguous version if the cache capacity is increased to 128 MB. This makes the

contiguous `ocean` code the SPLASH-2 program with the worst cache behavior on CMPs. Optimizations like that are used throughout the SPLASH-2 suite. In the case of `ocean` the two available versions made it possible to make a direct comparison, but an evaluation in all other cases is likely to require a rewrite of most of the SPLASH-2 suite. The high miss ratio is the reason why the contiguous version of `ocean` was identified as most similar to `canneal` by the clustering algorithm.

Intense Sharing More Common

Shared writes can be used as an approximation for the amount of communication between threads that takes place through a shared cache of a CMP. Figure 3.9 shows that the communication intensity is about the same for both suites. It is more concentrated in the case of SPLASH-2, where the two `lu` workloads are responsible for most of the shared writes within the suite. The large growth in shared writes when the cache capacity is increased from 4 MB to 8 MB is almost entirely caused by these two programs. Their ratios increase from 0.88% to 26.18% and from 2.80% to 27.40%. They remain on that level for all other cache sizes. This increase coincides with a drop in the miss rate from 0.11% to 0.01% in both cases when the cache becomes big enough to keep the shared part of the working set that is less frequently used by the programs. This unusual behavior is the reason why both `lu` programs have been identified as different from all other workloads by the redundancy analysis. The fact that the program is contained twice in two very similar versions artificially inflates the average shared write ratio of SPLASH-2. A larger number of PARSEC workloads show increased sharing activity.

Inclusion of Pipeline Model

Another difference between PARSEC and SPLASH-2 is the inclusion of workloads that employ the pipeline programming model in PARSEC. The programs `dedup` and `ferret` use pipelines with functional decomposition, i.e., the various pipeline stages have their own specialized thread pools that handle all the work for their assigned pipeline stage. Unlike in the case of workloads such as HPC programs, the threads of these programs execute different parts of the code. This programming model is frequently used to develop commercial workloads because it allows to break down the problem into independent

tasks that can be assigned to different development teams, resulting in lower overall complexity and development cost. As the data streams through the pipeline, it is handed from thread to thread, which perform different operations on the data until the overall goal of the program has been accomplished. A consequence of this model is that in general all communication between the pipeline stages is also communication between different cores. In extreme cases this can be as much as all of the input data, making efficient communication channels with high bandwidth between cores a necessity. Shared caches of CMPs satisfy these requirements. Figure 3.9 shows that both `dedup` and `ferret` make use of this aspect of CMPs.

3.7.2 Data Growth

World data is currently doubling every three years [24]. This trend is expected to accelerate further. With this huge amount of information comes the need to process and understand it. An example for a class of programs that deal with this vast amount of data are RMS programs. These workloads employ models that allow them to have a basic understanding of the data they process [24]. For example, the `bodytrack` program employs a model of the human body to detect a person being shown in multiple video streams.

The use of models is nothing new for computing. What has changed is the order of magnitude of the data that must be handled. Both PARSEC and SPLASH-2 contain programs that employ models, but only the algorithms and inputs of PARSEC workloads capture the large increase of data volume that is currently taking place. The compressed archive that contains the whole suite with all inputs is 16 MB in the case of SPLASH-2. For PARSEC, it is 2.7 GB. How does this affect workloads?

Large Working Sets More Common

Larger inputs are likely to result in larger working sets or require streaming program behavior. Figure 3.8 shows the miss rates of PARSEC and SPLASH-2 workloads. For smaller caches PARSEC workloads have a significantly higher average miss rate. The difference is 0.26% for a 1 MB cache, approximately one fourth more. It decreases to 0.11% for an 8 MB cache. SPLASH-2 workloads have an average miss rate 0.02% higher than PARSEC workloads if 16 MB caches are used. This trend continues to the end of the spectrum of cache sizes.

A closer look reveals that most of the SPLASH-2 misses are caused by only one program, the contiguous version of `ocean`. The last section already explained that this behavior is the consequence of optimizations for distributed shared memory machines that negatively affect the program on CMPs. The contiguous version of `ocean` should not be used on CMPs in favor of the non-contiguous version. This makes `fft` the only program with a noticeably higher miss rate. Other SPLASH-2 programs have miss rates that are within a very narrow range on a lower level. PARSEC captures a greater range of miss rates in comparison. It includes workloads such as `blackscholes`, which has the lowest miss rate of all programs (0.01% with 4 MB caches), up to `canneal`, which has the worst cache behavior of all benchmarks (miss rate of 3.18% with 4 MB caches). A total of four of its programs have a noticeably high miss rate (`canneal`, `facesim`, `ferret` and `streamcluster`).

Both PARSEC and SPLASH-2 contain workloads that can generate their own input. Can this feature be used to generate inputs for SPLASH-2 benchmarks that have large working sets comparable to PARSEC workloads? Unfortunately, this is not the case in practice. Most SPLASH-2 codes have runtime requirements that grow superlinearly with the size of the input, whereas its working sets grow no more than linearly in most cases [96]. Time limitations will thus constrain how large input problems can be in practice. For this study, the SPLASH-2 inputs were already sized such that the runtime of all benchmarks was about the same.

Table 3.3 summarizes how time and space requirements grow with the input size for all SPLASH-2 programs that have a parameter to scale their inputs. Only `fmm`, `radix` and the spatial version of `water` employ algorithms for which the computational requirements grow no faster than the memory requirements. `Barnes` can also be included in this list for practical purposes since the $\log N$ factor can be considered constant. This limits the number of workloads that can be tuned to reflect the enormous growth of data to only four, too few for most scientific studies. Computationally intensive workloads with relatively small working sets are a characteristic of the SPLASH-2 suite.

3.8 Partial Use of PARSEC

It has been shown that an incorrect benchmark subset selection can lead to misleading results [19]. This section uses redundancy analysis to suggest possible subsets of PAR-

Code	Time	Space
barnes	$N \log N$	N
fft	$N^{1.5} \log N$	N
fmm	N	N
lu (contig.)	N^3	N
lu (non-contig.)	N^3	N
ocean (contig.)	N^3	N^2
ocean (non-contig.)	N^3	N^2
radix	N	N
water (nsquared)	N^2	N
water (spatial)	N	N

Table 3.3: Growth rate of time and memory requirements of the SPLASH-2 workloads that have a parameter to scale the input size N. In most cases execution time grows much faster than the working set size.

SEC that can be used for microarchitectural studies. This approach can lead to optimal subsets [100]. Figure 3.10 shows the dendrogram which is obtained by clustering only the PARSEC workloads. It is important to point out that it cannot be compared directly to the dendrogram shown in Figure 3.1. Only a subset of the available data was used for Figure 3.10 because the SPLASH-2 workloads were not included. The PCA weighs all characteristics to form the linear combinations which become the PCs. Any changes to the data will in general result in different weighs. However, a direct comparison shows that the overall properties and similarities of the programs are the same in both cases.

The dendrogram can be used to pick a subset of the available workloads which minimal overlap as follows: Starting on the right, a vertical line can be moved towards the left side of the chart. As the line moves left it will intersect an increasing number of branches of the dendrogram, each one representing a cluster containing at least one PARSEC workload. As soon as the number of intersections equals the desired number of programs, exactly one workload can be selected from each cluster. The resulting selection will have a minimal amount of overlap of the characteristics which were used for this study. For example, a good subset of six programs would include `swaptions`, `canneal`, `x264`, `streamcluster`, `bodytrack` and one of the remaining workloads.

A subset obtained using the presented dendrogram should only be understood as a suggestion. Depending on the study, a different set of characteristics might be more appro-

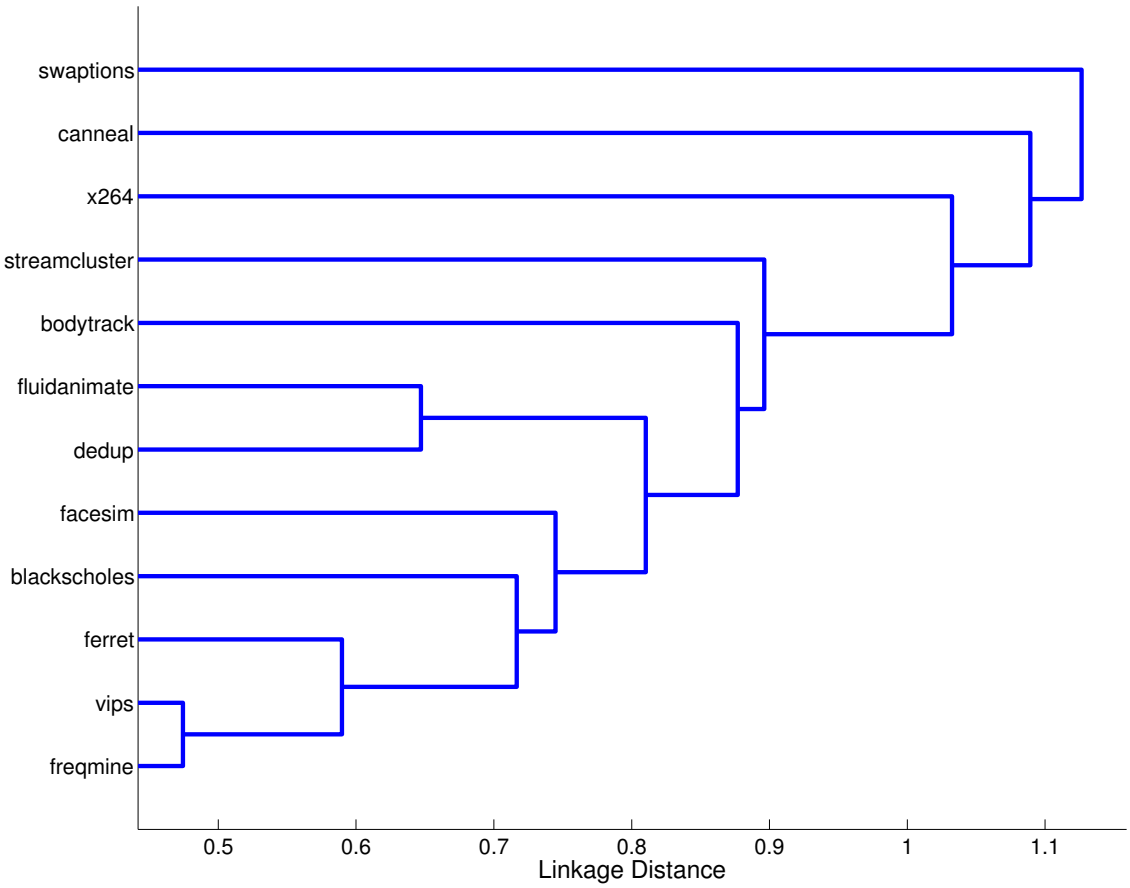

Figure 3.10: Redundancy within the PARSEC suite. The dendrogram can be used to choose a subset of PARSEC with minimal overlap.

priate, which would result in a different dendrogram. The characterization in Chapter 4 can be used as a basis to make an informed decision.

3.9 Related Work

Statistical analysis of benchmark characteristics with PCA and hierarchical clustering is a commonly used method. Eeckhout et al. were first to make use of it for workload analysis [51]. Giladi et al. analyzed the redundancy within the SPEC CPU89 suite [28]. They show that a subset of only six programs is sufficient to capture most of the diversity of the SPEC CPU89 suite. Vandierendonck et al. came to a similar conclusion when they studied the SPEC CPU2000 suite on 340 different machines [31]. Phansalkar et al. presented

a study of redundancy and application balance of SPEC CPU2006 on five different machines using six characteristics [73]. Hoste et al. proposed the use of genetic algorithms for comparisons of benchmarks [43]. Their approach is able to reduce the number of characteristics that have to be measured. Joshi et al. compared SPEC benchmarks across generations of the suite [42]. The selection of characteristics of this study differs from previous work because its focus is on the parallel behavior of programs. It therefore uses shared-memory characteristics.

Previous work on pipelining can be grouped into three categories: Studies that deal with pipelining as a method to manually parallelize existing programs, pipelining as a new programming model and pipelining as used by compilers to automatically parallelize serial programs.

Pipelining has gained popularity as a method to manually parallelize existing programs. This is a relatively recent trend: Kuck published a comprehensive survey about parallel architectures and programming models [48] over thirty years ago. He covers various early methods to parallelize programs but does not include the pipeline model. At that time multiprocessor machines were typically used to solve scientific problems which lend themselves well to domain decompositioning for parallelization purposes. When parallel architectures became popular for enterprise computing, different methods were needed and pipelining started to get used as a method to parallelize programs from that domain. One type of workload from that domain is main memory transaction processing. Li and Naughton studied the use of pipelining in those programs [54]. They demonstrate that pipelined programs on multiprocessors can achieve higher throughput and less locking overhead. Subhlok et al. study how the stages of a pipeline can be mapped optimally to processors [87]. They developed a new algorithm to compute a mapping that optimizes the latency with respect to constraint throughput and vice versa. The algorithm addresses the general mapping problem, which includes processor assignment, clustering and replication. As the popularity of pipelining grew and its benefits became more widely known, researchers became interested in its application to other types of programs. Thies et al. presented a systematic technique to parallelize streaming applications written in C with the pipeline parallelization model [90]. They suggest a set of annotations that programmers can use to parallelize legacy C programs so they can take advantage of shared-memory multiprocessors. The programmer is assisted by a dynamic analysis that traces the communication of memory locations at runtime.

The success of pipelining inspired a new type of programming model - the *stream programming model*. Stream programming is a parallelization approach that decomposes a program into a parallel network of specialized kernels which are then mapped to processing elements [18, 29, 46]. Data is organized as streams, which is a sequence of similar elements. A kernel in the stream programming model consumes streams, performs a computation, and produces a set of output streams. It corresponds to a pipeline stage of the pipeline programming model. Stream programs are suitable for execution on general-purpose multiprocessors [49, 56].

As it became clear that almost any type of program can be expressed as a pipeline the use of pipelining to automatically parallelize serial programs became a focus of attention. One such method is *Decoupled Software Pipelining* (DSWP). DSWP is an automatic parallelization method which uses the pipeline model [72, 78]. It exploits the fine-grained pipeline parallelism inherent in most applications to create a multithreaded version of the program that implements a parallel pipeline. Low-overhead synchronization between the pipeline stages can be implemented with a special synchronization array [78].

3.10 Conclusions

This chapter statistically compared program characteristics that capture instruction mix, communication and memory behavior of the PARSEC and SPLASH-2 benchmark suites on chip multiprocessors. The redundancy analysis showed that PARSEC and SPLASH-2 are composed of programs with fundamentally different properties. No single reason for the differences could be identified, but some important factors are differences in the instruction mix, the cache sharing intensity and the working set sizes. PARSEC is the more diverse suite in direct comparison.

Some of the observed differences can be explained by the inclusion of the pipeline model in PARSEC. Workloads which make use of the pipeline parallelization approach seem to have different characteristics. Other causes might be the proliferation of CMPs, which have made parallel computing a widely available commodity. Multiprocessors are now used in program domains that have little in common with High-Performance Computing. SPLASH-2 workloads, however, are optimized for distributed shared memory machines. As the example of `ocean` showed this can alter the observed characteristics to the point where the optimizations have more impact on the program behavior on CMPs

than the actual algorithm. The enormous growth of data has lead to an inflation of working set sizes. Input size and processing time limitations prevent that this behavior can be adequately captured with the SPLASH-2 suite.

So which one is the ”right” suite to use? It depends. SPLASH-2 remains an important and useful program collection. Researchers should consider what type of programs are most relevant for their study when they decide on a selection of benchmark programs. The presented comparison allows them to understand the implications of their choice. Whatever their decision, scientists should expect different results.

Chapter 4

Characterization of PARSEC

4.1 Introduction

The purpose of benchmarking for research is insight, not numbers. This means that results obtained with a benchmark program must be explainable, which requires a thorough understanding of the machine and the workload. The goal of PARSEC is to study parallel shared-memory machines. The following characteristics of a workload are therefore of particular interest and will be discussed in this chapter:

Parallelization PARSEC benchmarks use different parallel models which have to be analyzed in order to know whether the programs can scale well enough for the analysis of CMPs of a certain size.

Working sets and locality Knowledge of the cache requirements of a workload are necessary to identify benchmarks suitable for the study of CMP memory hierarchies.

Communication-to-computation ratio and sharing The communication patterns of a program determine the potential impact of private caches and the on-chip network on performance.

Off-chip traffic The off-chip traffic requirements of a program are important to understand how off-chip bandwidth limitations of a CMP can affect performance.

This chapter makes the following contributions:

- The presented characterization of the PARSEC workloads allows other researchers to interpret the results which they have obtained with the benchmark suite.
- The identified requirements allow us to sketch out the architectural properties that future multiprocessor machines must have.

The work presented in this chapter was previously published in [11].

4.2 Methodology

This section explains how the PARSEC benchmark suite was characterized. In order to characterize all applications, several trade-off decisions are made. Given a limited amount of computational resources, higher accuracy comes at the expense of a lower number of experiments. The employed methodology follows the approach of similar studies [41, 96] and chose faster but less accurate execution-driven simulation to characterize the PARSEC workloads. This approach is feasible because this study is limited to fundamental program properties which should have a high degree of independence from architectural details. Where possible, measurement results from real machines are supplied. This methodology allows the collection of the large amounts of data necessary for this study. Machine models comparable to real processors were preferred over unrealistic models which might have been a better match for the program needs.

4.2.1 Experimental Setup

CMP$im [41] was used for the workload characterization. CMP$im is a plug-in for Pin [58] that simulates the cache hierarchy of a CMP. Pin is similar to the ATOM toolkit [86] for Compaq's Tru64 Unix on Alpha processors. It uses dynamic binary instrumentation to insert routines at arbitrary points in the instruction stream. For the characterization a single-level cache hierarchy of a CMP was simulated and its parameters varied. The baseline cache configuration was a shared 4-way associative cache with 4 MB capacity and 64 byte lines. By default the workloads used 8 cores. All experiments were conducted on a set of Symmetric Multiprocessor (SMP) machines with x86 processors and Linux. The programs were compiled with `gcc` 4.2.1.

Because of the large computational cost simulations with the `native` input set could be performed, instead the `simlarge` inputs were used for all simulations. Any known differences between the two sets are described qualitatively.

4.2.2 Methodological Limitations and Error Margins

For their characterization of the SPLASH-2 benchmark suite, Woo et al. fixed a timing model which they used for all experiments [96]. They give two reasons: First, nondeterministic programs would otherwise be difficult to compare because different execution paths could be taken, and second, the characteristics they study are largely independent from an architecture. They also state that they believe that the timing model should have only a small impact on the results. While I used similar characteristics and share this belief, I think a characterization study of multithreaded programs should nevertheless analyze the impact of nondeterminism on the reported data. Furthermore, because the used methodology is based on execution on real machines combined with dynamic binary instrumentation, it can introduce additional latencies, and a potential concern is that the nondeterministic thread schedule is altered in a way that might affect the reported results in unpredictable ways. A sensitivity analysis was therefore conducted to quantify the impact of nondeterminism.

Alameldeen and Wood studied the variability of nondeterministic programs in more detail and showed that even small pseudo-random perturbations of memory latencies are effective to force alternate execution paths [2]. I adopted their approach and modified CMP$im to add extra delays to its analysis functions. Because running all experiments multiple times as Alameldeen and Wood did would be prohibitively expensive, a random subset of all experiments for each metric which was use was selected instead and its error margins reported here.

The measured quantities deviated by no more than $\pm 0.04\%$ from the average, with the following two exceptions. The first excpetion is metrics of data sharing. In two cases (`bodytrack` and `swaptions`) the classification is noticeably affected by the nondeterminism of the program. This is partially caused because shared and thread-private data contend aggressively for a limited amount of cache capacity. The high frequency of evictions made it difficult to classify lines and accesses as shared or private. In these cases, the maximum deviation of the number of accesses from the average was as high as

$\pm 4.71\%$, and the amount of sharing deviated by as much as $\pm 15.22\%$. This uncertainty was considered for this study and no conclusions were drawn where the variation of the measurements did not allow it. The second case of high variability is when the value of the measured quantity is very low (below 0.1% miss rate or corresponding ratio). In these cases the nondeterministic noise made measurements difficult. I do not consider this a problem because this study focuses on trends of ratios, and quantities that small do not have a noticeable impact. It is however an issue for the analysis of working sets if the miss rate falls below this threshold and continues to decrease slowly. Only few programs are affected, and the given estimate of their working set sizes might be slightly off in these cases. This is primarily an issue inherent to experimental working set analysis, since it requires well-defined points of inflection for conclusive results. Moreover, in these cases the working set sizes seem to vary nondeterministically, and researchers should expect slight variations for each benchmark run.

The implications of these results are twofold: First, they show that the employed methodology is not susceptible to the nondeterministic effects of multithreaded programs in a way that might invalidate the reported findings. Second, they also confirm that the metrics which are presented in this chapter are fundamental program properties which cannot be distorted easily. The reported application characteristics are likely to be preserved on a large range of architectures.

4.3 Parallelization

This section discusses the parallelization of the PARSEC suite. As will be demonstrated in Section 4.4, several PARSEC benchmarks (`canneal`, `dedup`, `ferret` and `freqmine`) have working sets so large they should be considered unbounded for an analysis. These working sets are only limited by the amount of main memory in practice and they are actively used for inter-thread communication. The inability to use caches efficiently is a fundamental property of these program and affects their concurrent behavior. Furthermore, `dedup` and `ferret` use a complex, heterogeneous parallelization model in which specialized threads execute different functions with different characteristics at the same time. These programs employ a pipeline with dedicated thread pools for each parallelized pipeline stage. Each thread pool has enough threads to occupy the whole CMP, and it is the responsibility of the scheduler to assign cores to threads in a manner that maximizes

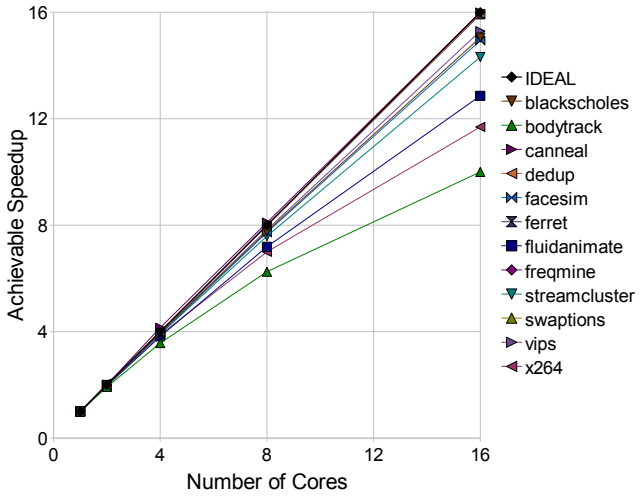

Figure 4.1: Upper bound for speedup of PARSEC workloads based on instruction count.

the overall throughput of the pipeline. Over time, the number of threads active for each stage will converge against the inverse throughput ratios of the individual pipeline stages relative to each other.

Woo et al. use an abstract machine model with a uniform instruction latency of one cycle to measure the speedups of the SPLASH-2 programs [96]. They justify their approach by pointing out that the impact of the timing model on the characteristics which they measure - including speedup - is likely to be low. Unfortunately, this is not true in general for PARSEC workloads. While Section 4.2.2 explains that the fundamental program properties such as miss rate and instruction count are largely not susceptible to timing shocks, the synchronization and timing behavior of the programs is. Using a timing model with perfect caches significantly alters the behavior of programs with unbounded working sets, for example how long locks to large, shared data structures are held. Moreover, any changes of the timing model have a strong impact on the number of active threads of programs which employ thread specialization. It will thus affect the load balance and synchronization behavior of these workloads. I believe it is not possible to discuss the timing behavior of these programs without also considering for example different schedulers, which is beyond the scope of this work. Similar dependencies of commercial workloads on their environment are already known [1, 8].

Unlike Woo et al. who measured *actual* concurrency on an abstract machine, this section therefore analyzes *inherent* concurrency and its limitations. The employed approach

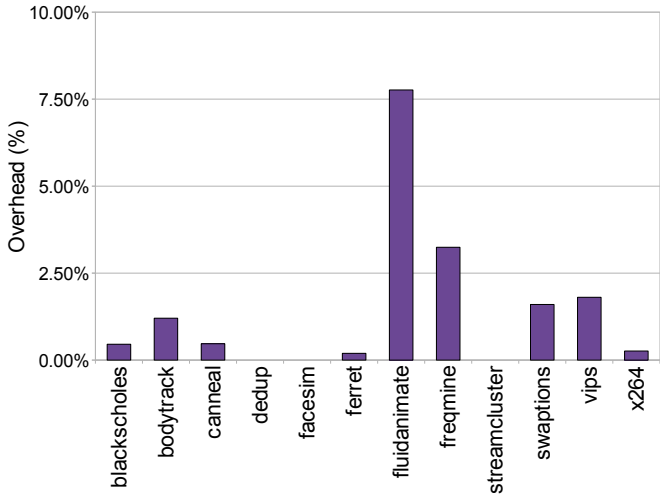

Figure 4.2: Parallelization overhead of PARSEC benchmarks. The chart shows the slowdown of the parallel version on one core over the serial version.

is based on the number of executed instructions in parallel and serial regions of the code. Any delays due to blocking on contended locks and load imbalance are neglected. This methodology is feasible because performance is not studied, the focus of this chapter is on fundamental program characteristics. The presented data is largely timing-independent and a suitable measure of the concurrency inherent in a workload.

The results in Figure 4.1 show the maximum achievable speedup measured that way. The numbers account for limitations such as unparallelized code sections, synchronization overhead and redundant computations. PARSEC workloads can achieve *actual* speedups close to the presented numbers. During the development of the programs it was verified on a large range of architectures that lock contention and other timing-dependent factors are not limiting factors, but there is no known way to show it in a platform-independent way given the complications outlined above.

The maximum speedup of `bodytrack`, `x264` and `streamcluster` is limited by serial sections of the code. `Fluidanimate` is primarily limited by growing parallelization overhead. On real machines, `x264` is furthermore bound by a data dependency between threads, however this has only a noticeable impact on machines larger than the ones described here. It is recommended to run `x264` with more threads than cores, since modeling and exposing these dependencies to the scheduler is a fundamental aspect of its parallel algorithm, comparable to the parallel algorithms of `dedup` and `ferret`. Figure 4.2 shows the slowdown of the parallel version on 1 core over the serial version. The numbers show

that all workloads use efficient parallel algorithms which are not substantially slower than the corresponding serial algorithms.

PARSEC programs scale well enough to study CMPs. I believe they are also useful on machines larger than the ones analyzed here. The PARSEC suite exhibits a wider variety of parallelization models than previous benchmark suites such as the pipeline model. Some of its workloads can adapt to different timing models and can use threads to hide latencies. It is important to analyze these programs in the context of the whole system.

4.4 Working Sets and Locality

The temporal locality of a program can be estimated by analyzing how the miss rate of a processor's cache changes as its capacity is varied. Often the miss rate does not decrease continuously as the size of a cache is increased, but stays on a certain level and then makes a sudden jump to a lower level when the capacity becomes large enough to hold the next important data structure. For CMPs an efficient functioning of the last cache level on the chip is crucial because a miss in the last level will require an access to off-chip memory.

To analyze the working sets of the PARSEC workloads a cache shared by all processors was studied. The results are presented in Figure 4.3. Table 4.1 summarizes the important characteristics of the identified working sets. Most workloads exhibit well-defined working sets with clearly identifiable points of inflection. Compared to SPLASH-2, PARSEC working sets are significantly larger and can reach hundreds of megabytes such as in the cases of `canneal` and `freqmine`.

Two types of workloads can be distinguished: The first group contains benchmarks such as `bodytrack` and `swaptions` which have working sets no larger than 16 MB. These workloads have a limited need for caches with a bigger capacity, and the latest generation of CMPs often already has caches sufficiently large to accommodate most of their working sets. The second group of workloads is composed of the benchmarks `canneal`, `ferret`, `facesim`, `fluidanimate` and `freqmine`. These programs have very large working sets of sizes 65 MB and more, and even with a relatively constrained input set such as `simlarge`, their working sets can reach hundreds of megabytes. Moreover, the need of those workloads for cache capacity is nearly insatiable and grows with the amount of data which they process. Table 4.1 gives estimates for the largest working set of each PARSEC workload for the `native` input set. In several cases they are significantly larger

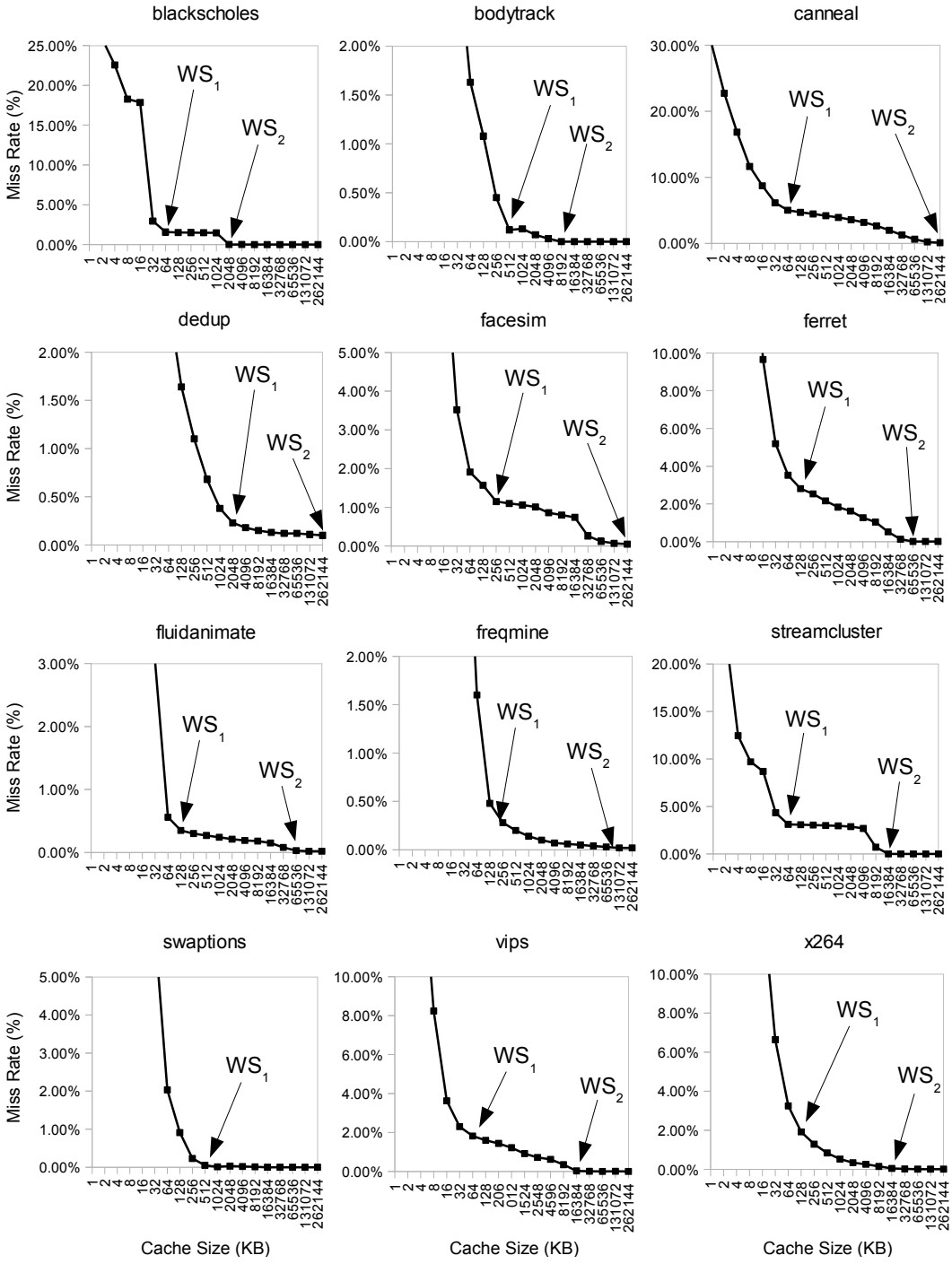

Figure 4.3: Miss rates for various cache sizes. The data assumes a shared 4-way associative cache with 64 byte lines. WS_1 and WS_2 refer to important working sets which are analyzed in more detail in Table 4.1. The cache requirements of PARSEC benchmark programs can reach hundreds of megabytes.

Program	Input Set simlarge						Input Set native
	Working Set 1			Working Set 2			Working Set 2
	Data Structure	Size	Growth Rate	Data Structure	Size	Growth Rate	Size Estimate
blackscholes	options	64 KB	C	portfolio data	2 MB	C	same
bodytrack	edge maps	512 KB	const.	input frames	8 MB	const.	same
canneal	elements	64 KB	C	netlist	256 MB	DS	2 GB
dedup	data chunks	2 MB	C	hash table	256 MB	DS	2 GB
facesim	tetrahedra	256 KB	C	face mesh	256 MB	DS	same
ferret	images	128 KB	C	data base	64 MB	DS	128 MB
fluidanimate	cells	128 KB	C	particle data	64 MB	DS	128 MB
freqmine	transactions	256 KB	C	FP-tree	128 MB	DS	1 GB
streamcluster	data points	64 KB	C	data block	16 MB	user-def.	256 MB
swaptions	swaptions	512 KB	C	same as WS_1	same	same	same
vips	image data	64 KB	C	image data	16 MB	C	same
x264	macroblocks	128 KB	C	reference frames	16 MB	C	same

Table 4.1: Important working sets and their growth rates. DS represents the data set size and C is the number of cores. The working set sizes are taken from Figure 4.3. The values for the native input set are analytically derived estimates. Working sets that grow proportional to the number of cores C are aggregated private working sets and can be split up to fit into correspondingly smaller, private caches.

and can even reach gigabytes. These large working sets are often the consequence of an algorithm that operates on large amounts of collected input data. Ferret for example keeps a data base of feature vectors of images in memory to find the images most similar to a given query image. The cache and memory needs of these applications should be considered unbounded, as they become more useful to their users if they can work with increased amounts of data. Programs with unbounded working sets are canneal, dedup, ferret and freqmine.

Figure 4.4 presents an analysis of the spatial locality of the PARSEC workloads. The data shows how the miss rate of a shared cache changes with line size. All programs benefit from larger cache lines, but to different extents. Facesim, fluidanimate and streamcluster show the greatest improvement as the line size is increased, up to the the maximum value of 256 bytes which was used. These programs have streaming behavior, and an increased line size has a prefetching effect which these workloads can take advantage of. Facesim for example spends most of its time updating the position-based state of the model, for which it employs an iterative Newton-Raphson algorithm. The algorithm iterates over the elements of a sparse matrix which is stored in two one-

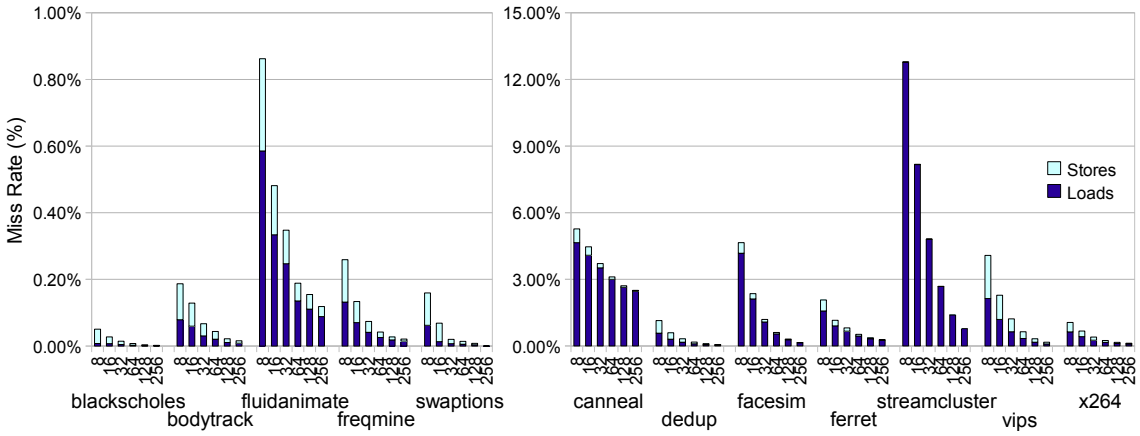

Figure 4.4: Miss rates as a function of line size. The data assumes 8 cores sharing a 4-way associative cache with 4 MB capacity. Miss rates are broken down to show the effect of loads and stores.

dimensional arrays, resulting in a streaming behavior. All other programs also show good improvement of the miss rate with larger cache lines, but only up to line sizes of about 128 bytes. The miss rate is not substantially reduced with larger lines. This is due to a limited size of the basic data structures employed by the programs. They represent independent logical units, each of which is intensely worked with during a computational phase. For example, `x264` operates on macroblocks of 8×8 pixels at a time, which limits the sizes of the used data structures. Processing a macroblock is computationally intensive and largely independent from other macroblocks. Consequently, the amount of spatial locality is bounded in these cases.

The rest of this analysis chooses a cache capacity of 4 MB for all experiments. A matching cache size for each workload could have been used, but that would have made comparisons very difficult, and the use of very small or very large cache sizes is not realistic. Moreover, in the case of the workloads with an unbounded working set size, a working set which completely fits into a cache would be an artifact of the limited simulation input size and would not reflect realistic program behavior.

4.5 Communication-to-Computation Ratio and Sharing

This section discusses how PARSEC workloads use caches to communicate. Most PARSEC benchmarks share data intensely. Two degrees of sharing can be distinguished:

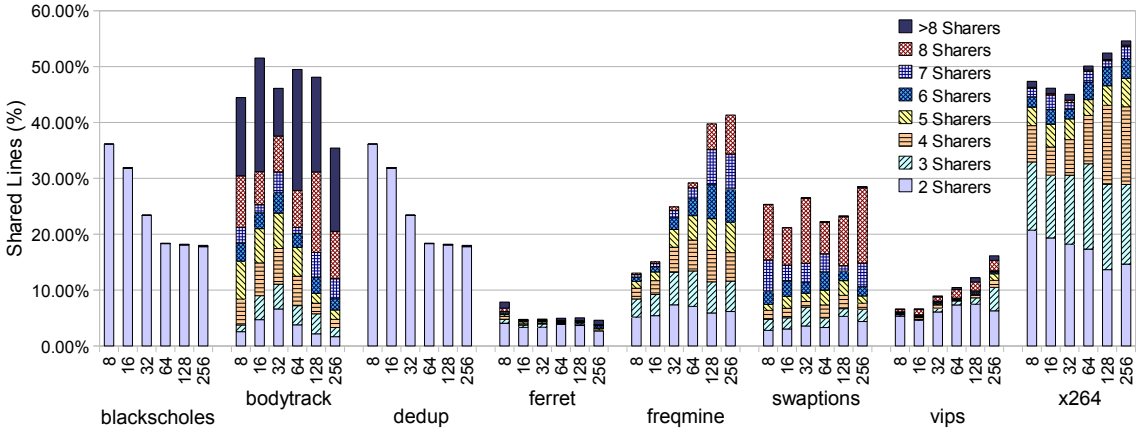

Figure 4.5: Portion of a 4-way associative cache with 4 MB capacity which is shared by 8 cores. The line size is varied from 8 to 256 bytes.

Shared data can be read-only during the parallel phase, in which case it is only used for lookups and analysis. Input data is frequently used in such a way. But shared data can also be used for communication between threads, in which case it is also modified during the parallel phase. Figure 4.5 shows how the line size affects sharing. The data combines the effects of false sharing and the access pattern of the program due to constrained cache capacity. Figure 4.6 illustrates how the programs use their data. The chart shows what data is accessed and how intensely it is used. The information is broken down in two orthogonal ways, resulting in four possible types of accesses: Read and write accesses and accesses to thread-private and shared data. Additionally, the amount of true shared accesses is given. An access is a true access if the last reference to that line came from another thread. True sharing does not count repeated accesses by the same thread. It is a useful metric to estimate the requirements for the cache coherence mechanism of a CMP: A true shared write can trigger a coherence invalidate or update, and a true shared read might require the replication of data. All programs exhibit very few true shared writes.

Four programs (`canneal`, `facesim`, `fluidanimate` and `streamcluster`) showed only trivial amounts of sharing. They have therefore not been included in Figure 4.5. In the case of `canneal`, this is a result of the small cache capacity. Most of its large working set is shared and actively worked with by all threads. However, only a minuscule fraction of it fits into the cache, and the probability that a line is accessed by more than one thread before it gets replaced is very small in practice. With a 256 MB cache, 58% of its cached data is shared. `Blackscholes` shows a substantial amount of sharing,

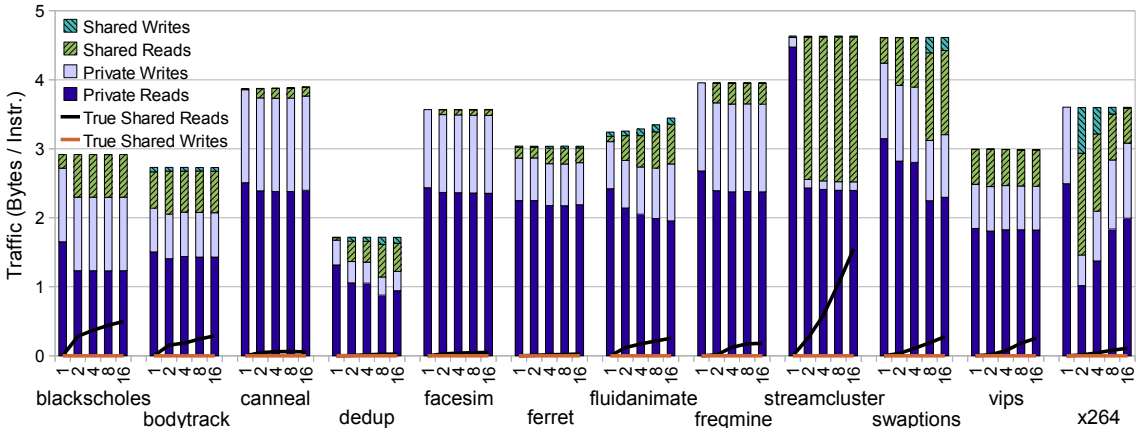

Figure 4.6: Traffic from cache in bytes per instruction for 1 to 16 cores. The data assumes a shared 4-way associative cache with 64 byte lines.

but almost all its shared data is only accessed by two threads. This is a side-effect of the parallelization model: At the beginning of the program, the boss threads initializes the portfolio data before it spawns worker threads which process parts of it in a data-parallel way. As such, the entire portfolio is shared between the boss thread and its workers, but the worker threads can process the options independently from each other and do not have to communicate with each other. `Ferret` shows a modest amount of data sharing. Like the sharing behavior of `canneal`, this is caused by severely constrained cache capacity. `Ferret` uses a database that is scanned by all threads to find entries similar to the query image. However, the size of the database is practically unbounded, and because threads do not coordinate their scans with each other it is unlikely that a cache line gets accessed more than once. `Bodytrack` and `freqmine` exhibit substantial amounts of sharing due to the fact that threads process the same data. The strong increase of sharing of `freqmine` is caused by false sharing, as the program uses an array-based tree as its main data structure. Larger cache lines will contain more nodes, increasing the chance that the line is accessed by multiple threads. `Vips` has some shared data which is mostly used by only two threads. This is also predominantly an effect of false sharing since image data is stored in a consecutive array which is processed in a data-parallel way by threads. `X264` uses significant amounts of shared data, most of which is only accessed by a low number of threads. This data is the reference frames, since a thread needs this information from other stages in order to encode the frame it was assigned. Similarly, the large amount of shared data of `dedup` is the input which is passed from stage to stage.

Most PARSEC workloads use a significant amount of communication, and in many cases the volume of traffic between threads can be so high that efficient data exchange via a shared cache is severely constrained by its capacity. An example for this is `x264`. Figure 4.6 shows a large amount of writes to shared data, but contrary to intuition its share diminishes rapidly as the number of cores is increased. This effect is caused by a growth of the working sets of `x264`: Table 4.1 shows that both working set WS_1 and WS_2 grow proportional to the number of cores. WS_1 is mostly composed of thread-private data and is the one which is used more intensely. WS_2 contains the reference frames and is used for inter-thread communication. As WS_1 grows, it starts to displace WS_2, and the threads are forced to communicate via main memory. Two more programs which communicate intensely are `dedup` and `ferret`. Both programs use the pipeline parallelization model with dedicated thread pools for each parallel stage, and all data has to be passed from stage to stage. `Fluidanimate` also shows a large amount of inter-thread communication, and its communication needs grow as the number of threads increase. This is caused by the spatial partitioning that `fluidanimate` uses to distribute the work to threads. Smaller partitions mean a worse surface to volume ratio, and communication grows with the surface.

Overall, most PARSEC workloads have complex sharing patterns and communicate actively. Pipelined programs can require a large amount of bandwidth between cores in order to communicate efficiently. Shared caches with insufficient capacity can limit the communication efficiency of workloads, since shared data structures might get displaced to memory.

4.6 Off-Chip Traffic

This section analyzes the off-chip bandwidth requirements of the PARSEC workloads. The goal is to understand how the traffic of an application grows as the number of cores of a CMP increases and how the memory wall will limit performance. A shared cache was simulated again and the development of the traffic analyzed as the number of cores increases. The results are presented in Figure 4.7.

The data shows that the off-chip bandwidth requirements of the `blackscholes` workload are small enough so that memory bandwidth is unlikely to be an issue. `Bodytrack`,

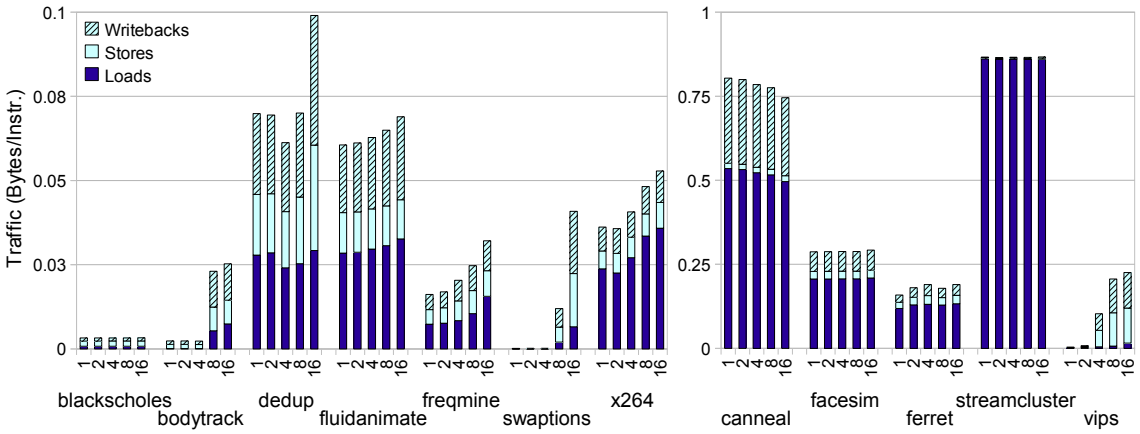

Figure 4.7: Breakdown of off-chip traffic for 1 to 16 cores. The data assumes a 4-way associative 4 MB cache with 64 byte lines, allocate-on-store and write-back policy.

`dedup`, `fluidanimate`, `freqmine`, `swaptions` and `x264` are more demanding. Moreover, these programs exhibit a growing bandwidth demand per instruction as the number of cores increases. In the case of `bodytrack`, most off-chip traffic happens in short, intense bursts since the off-chip communication predominantly takes place during the edge map computation. This phase is only a small part of the serial runtime, but on machines with constrained memory bandwidth it quickly becomes the limiting factor for scalability. The last group of programs is composed of `canneal`, `facesim`, `ferret`, `streamcluster` and `vips`. These programs have very high bandwidth requirements and also large working sets. `Canneal` shows a decreasing demand for data per instruction with more cores. This behavior is caused by improved data sharing.

It is important to point out that these numbers do not take the increasing instruction throughput of a CMP into account as its number of cores grows. A constant traffic amount in Figure 4.7 means that the bandwidth requirements of an application which scales linearly will grow exponentially. Since many PARSEC workloads have high bandwidth requirements and working sets which exceed conventional caches by far, off-chip bandwidth will be their most severe limitation of performance. Substantial architectural improvements are necessary to allow emerging workloads to take full advantage of larger CMPs.

4.7 Conclusions

The PARSEC benchmark suite is designed to provide parallel programs for the study for CMPs. PARSEC can be used to drive research efforts by application demands. It focuses on emerging desktop and server applications and does not have the limitations of other benchmark suites. It is diverse enough to be considered representative, it is not skewed towards HPC programs, it uses state-of-art algorithms and it supports research. In this study I characterized the PARSEC workloads to provide the basic understanding necessary to allow other researchers the effective use of PARSEC for their studies. I analyzed the parallelization, the working sets and locality, the communication-to-computation ratio and the off-chip bandwidth requirements of its workloads.

Chapter 5

Fidelity and Input Scaling

5.1 Introduction

Computer architects need detailed simulations for their microarchitecture designs. However, simulators are typically many orders of magnitude slower than a real machine. Running a set of benchmark applications with realistic inputs on a simulator will far exceed the design time limits, even using thousands of computers.

To reduce the number of simulated instructions of a benchmark, a commonly used method is to take a subset of the instructions with statistical sampling, which requires a sophisticated mechanism in the simulator [17, 20, 80, 97]. Another method is to reduce the size of the input for a benchmark [47, 91]. Reduced inputs are easy to use with any simulator, but there is no systematic framework for scaling input sets in a continuous way. Furthermore, there is a lack of understanding of the trade-offs between accuracy and cost involved when reducing inputs for a given benchmark.

A good benchmark suite should provide users with inputs of multiple levels of fidelity, scaling from realistic inputs for execution on real machines down to small inputs for detailed simulations. The key question is how to scale such inputs for a benchmark so that its execution with a scaled-down input produces meaningful performance predictions for computer architects.

This chapter presents a framework for the input-scaling problem of a benchmark. Scaled inputs are viewed as approximations of the original, full-sized inputs. As the inputs are scaled down, the benchmark becomes increasingly inaccurate. The question of how to create many derivatives of real inputs with varying size and accuracy motivates the

following optimization problem: *Given a time budget, what is the optimal selection of inputs for a set of benchmark programs?* This work shows that this optimization problem is the classical *multiple-choice knapsack problem* (MCKP), which is NP-hard. The formulation of the problem allows one to derive the general guidelines for designing multiple inputs with different levels of fidelity for a benchmark suite.

The chapter proposes the methodology and implementation of scaled inputs for the Princeton Application Repository for Shared-Memory Computers (PARSEC) [11]. The PARSEC benchmark suite is designed to represent emerging workloads. The current release (version 2.1) consists of 13 multithreaded programs in computer vision, video encoding, physical modeling, financial analytics, content-based search, and data deduplication. The methodology was used to design six sets of inputs with different levels of fidelity. The first version of PARSEC was released only two years ago. It has been well adopted by the computer architecture community to evaluate multicore designs and multiprocessor systems.

To evaluate the impact of input scaling, a measure called *approximation error* was defined. Using this measure, several input sets of PARSEC were analyzed to show which reduced inputs are more likely to distort the original characteristics of the PARSEC benchmarks. This chapter furthermore analytically derive a scope for each input which defines the range of architectures for which the input can be expected to be reasonably accurate. Such results are helpful for PARSEC users to understand the implications when creating customized inputs for their simulation time budgets.

This chapter makes several contributions. First, it presents a novel methodology to analyze how scaled inputs affect the accuracy of a benchmark suite. Second, it formulates the input selection problem for a benchmark as an optimization problem that maximizes the accuracy of a benchmark subject to a time constraint. Third, it describes how the PARSEC inputs have been scaled and shows which parts of the inputs are more likely to distort the original program characteristics. More importantly, it discusses in which situations small input sets might produce highly misleading program behavior. Finally, the chapter provides guidelines for users to create their own customized input sets for PARSEC.

The work presented in this chapter was previously published in [14].

5.2 Input Fidelity

This section introduces the conceptual foundation and terminology to discuss the inaccuracies in benchmark inputs. My concepts and definitions follow the terms commonly used in the fields of mathematical approximation theory and scientific modeling [4].

The main insight is that inputs for benchmark programs are almost never real program inputs. An example is a server program whose behavior is input dependent, but its input data contains confidential user information that cannot be made publicly available. Also, computationally intensive applications require too much time to complete. For that reason benchmark inputs are typically derived from real program inputs by reducing them in a suitable way. I call the process of creating a range of reduced inputs *input scaling*. This process can be compared to stratified sampling. The idea of stratified sampling is to take samples in a way that leverages knowledge about the sampling population so that the overall characteristics of the population are preserved as much as possible.

Benchmark inputs can be thought of as models of real inputs. Their purpose is to approximate real program behavior as closely as possible, but even if the benchmark program itself is identical to the actual application, some deviations from its real-world behavior must be expected. I call these deviations the *approximation error* because they are unintended side effects which can distort performance measurements in subtle ways. Sometimes very noticeable errors can be identified. I will refer to these errors as *scaling artifacts*. A scaling artifact can increase the inaccuracy of a benchmark and sometimes it may lead to highly misleading results. In such cases, scaling artifacts are good indicators whether an input set can accurately approximate the real workload inputs. Identifying such artifacts is essential to determine the constraints of a benchmark input set.

I define *fidelity* as the degree to which a benchmark input set produces the same program behavior as the real input set. An input set with high fidelity has a small approximation error with few or no identifiable scaling artifacts, whereas an input set with low fidelity has a large approximation error and possibly many identifiable scaling artifacts. Input fidelity can be measured by quantifying the approximation error. It is common that approximations are optimized to achieve a high degree of fidelity in a limited target area, possibly at the expense of reduced accuracy in other areas. I call this target area of high fidelity the *scope* of an input. The scope can be thought of as a set of restrictions for a reduced input beyond which the input becomes very inaccurate. An example is an overall

reduction of work units in an input set, which will greatly reduce execution time, but limit the amount of CPUs that the program can stress simultaneously. Running the benchmark on CMPs with more cores than the input includes may lead to many noticeable scaling artifacts in the form of idle CPUs.

5.2.1 Optimal Input Selection

Input scaling can be used to create a continuum of approximations of a real-world input set with decreasing fidelity and instruction count. This allows benchmark users to trade benefit in the form of benchmarking accuracy for lower benchmarking cost as measured by execution or simulation time. This motivates the question what the optimal combination of inputs from a given selection of input approximations is that maximizes the accuracy of the benchmark subject to a time budget.

If the time budget is fixed and an optimal solution is desired, the problem assumes the structure of the multiple-choice knapsack problem (MCKP), which is a version of the binary knapsack problem with the addition of disjoint multiple-choice constraints [66, 82]. For a given selection of m benchmark programs with disjoint sets of inputs N_i, $i = 1, \ldots, m$, I will refer to the approximation error of each respective input $j \in N_i$ with the variable $a_{ij} \geqq 0$. If solution variable $x_{ij} \in \{0, 1\}$ is defined for each input that specifies whether an input has been selected then the goal of the optimization problem can be formally defined as follows:

$$\min \sum_{i=1}^{m} \sum_{j \in N_i} a_{ij} x_{ij} \tag{5.1}$$

The solution of the problem is the set of all x_{ij} that describes which inputs are optimal to take. If the time each input takes to execute is referred to by $t_{ij} \geqq 0$ and if the total amount of time available for benchmark runs is given by $T \geqq 0$ then the solutions which are acceptable can be described as follows:

$$\sum_{i=1}^{m} \sum_{j \in N_i} t_{ij} x_{ij} \leqq T \tag{5.2}$$

Lastly, it has to be stated that exactly one input must be selected for each benchmark program and that no partial or negative inputs are allowed:

$$\sum_{j \in N_i} x_{ij} = 1 \quad i = 1, \ldots, m \tag{5.3a}$$

$$x_{ij} \in \{0,1\} \quad j \in N_i, \quad i = 1, \ldots, m \tag{5.3b}$$

MCKP is an NP-hard problem, but it can be solved in pseudo-polynomial time through dynamic programming [25,74]. The most efficient algorithms currently known first solve the linear version of MCKP (LMCKP) that is obtained if the integrality constraint $x_{ij} \in \{0,1\}$ is relaxed to $0 \leqq x_{ij} \leqq 1$. LMCKP is a variant of of the fractional knapsack problem and can be solved in $O(n)$ time by a greedy algorithm. The initial feasible solution that is obtained that way is used as a starting point to solve the more restrictive MCKP version of the problem with a dynamic programming approach. It keeps refining the current solution in an enumerative fashion by adding new classes until an optimal solution has been found. Such an algorithm can solve even very large data instances within a fraction of a second in practice, as long as the input data is not strongly correlated [74].

5.2.2 Scaling Model

The most common way to express the asymptotic runtime behavior of a program is to use a function of a single parameter N, which is the total size of the input. This is too simplistic for real-world programs, which usually have inputs that are described by several parameters, each of which may significantly affect the behavior of the programs. This section presents a simple scaling model for reasoning about the effect of input parameters on program behavior.

In this simple model, the inputs of a program are grouped into two components: the linear component and the complex component. The linear component includes the parts that have a linear effect on the execution time of the program such as streamed data or the number of iterations of the outermost loop of the workload. The complex component includes the remaining parts of the input which can have any effect on the program. For example, a fully defined input for a program that operates on a video would be composed of a complex part that determines how exactly to process each frame and a linear part that determines how many frames to process.

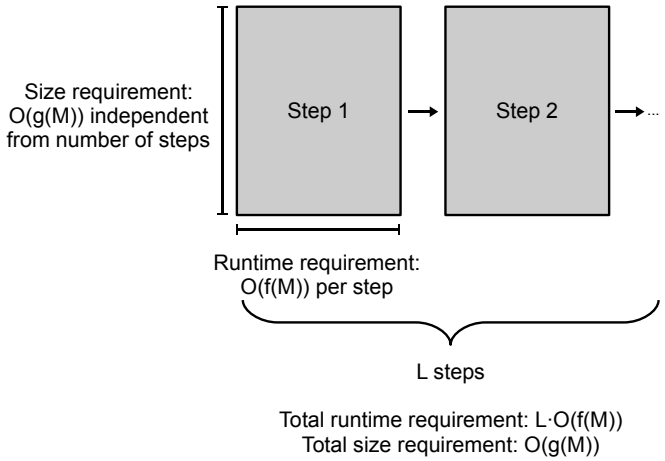

Figure 5.1: The impact that linear and complex input size scaling typically have on a program. Complex scaling of M usually affects the execution time $f(M)$ and the memory requirement $g(M)$. Linear scaling typically only changes the number of repetitions L.

Our motivation for this distinction is twofold: First, real-world workloads frequently apply complex computational steps which are difficult to analyze in a repetitive fashion that is easy to count and analyze. Second, the two components have different qualitative scaling characteristics because the complex input part often has to remain in a fairly narrow range whereas the linear component can often be chosen rather freely. In the example of the video processing program, the complex input part determining the work for each single frame is limited by the algorithms that are available and need to be within the range of video frame sizes that make sense in practice, which means there is a tight upper bound on the number of processing steps as well as lower and upper bounds on the frame resolution that make sense. These bounds significantly limit the scaling range, and the possible choice among different algorithms make this part of the input difficult to analyze. The number of frames, however, is unconstrained and can practically reach any number.

These two components of an input affect the input scaling behavior of the program in different ways, as summarized in Figure 5.1. The impact of the complex input parts is typically highly dependent on the workload so that few general guidelines can be given. It is common that the asymptotic runtime of the program increases superlinearly but its memory requirements often grow only linearly because most programs only keep the input data in a parsed but essentially unmodified form in memory. This property makes it hard to scale an input using its complex components without skewing it. Complex scaling

Input Set	Description	Time Budget	Purpose
`test`	Minimal execution time	N/A	Test & Development
`simdev`	Best-effort code coverage of real inputs	N/A	
`simsmall`	Small-scale experiments	$\lessapprox$ 1 s	Simulations
`simmedium`	Medium-scale experiments	$\lessapprox$ 4 s	
`simlarge`	Large-scale experiments	$\lessapprox$ 15 s	
`native`	Real-world behavior	$\lessapprox$ 15 min	Native execution

Table 5.1: The six standardized input sets offered by PARSEC listed in order of increasing size. Larger input sets guarantee the same properties of all smaller input sets. Time is approximate serial execution time on real machines.

almost always reduces working set sizes.

The linear input components typically do not affect the memory requirements or working set sizes of the program that much because they involve a form of repetition of previous steps. Their strong and direct impact on the execution time of the program make them suitable for input size scaling. Usually, there is no upper limit for linear input scaling, but reducing the input to the point where it includes few if any repetitions can often result in strong scaling artifacts because the individual steps are not exactly the same or not completely independent from each other. For example, if a program takes the output of its previous iteration as the input for its next iteration, it usually results in significant communication between threads. The underlying communication patterns may vary significantly from one iteration to the next. In this case, the number of repetitions included in the input must be large enough to stabilize what type of tasks the program performs on average.

5.3 PARSEC Inputs

This section describes the inputs of PARSEC and how they are scaled. PARSEC offers six input sets. Each set contains exactly one fully defined input for each PARSEC benchmark. An input is composed of all input files required by the program and a predetermined way to invoke the binary. Input sets can be distinguished by the amount of work their inputs contain. This determines what an input set can be used for. Table 5.1 gives an overview of the six PARSEC input sets ordered in ascending order by the allowed time budget.

The `native` input set is the closest approximations to realistic inputs, even though it is not authoritative. Only benchmarks with real-world programs using real-world inputs are

authoritative [35]. Smaller input sets can be considered increasingly inaccurate approximations of real-world inputs. Users of the benchmark suite should therefore generally use the largest input set possible. The smallest input set which I consider acceptable for at least some performance experiments is `simsmall`.

5.3.1 Scaling of PARSEC Inputs

PARSEC inputs predominantly use linear input scaling to achieve the large size reduction from real inputs to `native` and `simlarge` and a combination of linear and complex scaling to derive the `simmedium` and `simsmall` input sets from `simlarge`. For that reason the differences between real inputs, `native` and `simlarge` should be relatively small. The input sets `simdev` and `test` were created in a completely different way and should not be used for performance experiments at all. The various inputs suitable for performance measurements are summarized in Table 5.2.

Most parts of the complex input components are identical between the input sets `simlarge` and `native`. In seven cases at least one part of the complex input component is not identical between the two input sets: `Canneal`, `ferret`, `fluidanimate`, `freqmine`, `raytrace`, `streamcluster` and `x264` all have at least one input component that exhibits complex scaling behavior which might affect the program noticeably. However, only in the case of `streamcluster` could a noticeable and strong impact of that property on the program characteristics be measured. This is because there is a direct, linear relationship between the working set of the program and the selected block size, which can be freely chosen as part of the input and which has also been scaled between input sets. For `simlarge` the working set corresponding to the block size is 8 MB, which means a working set size between 64 MB and 128 MB can be expected for the `native` input. In all other cases the differences between `simlarge` and `native` can be expected to be negligible on contemporary machines.

`Blackscholes`, `dedup`, `swaptions` and `vips` all break their input into small chunks which are processed one after another. Their inputs are the easiest to scale and generally should show little variation. In the case of `blackscholes` and `dedup` some impact on the working set sizes should be expected because each input unit can be accessed more than once by the program.

Program	Input Set	Problem Size		Comments
		Complex Component	Linear Component	
blackscholes	simsmall		4,096 options	
	simmedium		16,384 options	
	simlarge		65,536 options	
	native		10,000,000 options	
bodytrack	simsmall	4 cameras, 1,000 particles, 5 layers	1 frame	
	simmedium	4 cameras, 2,000 particles, 5 layers	2 frames	
	simlarge	4 cameras, 4,000 particles, 5 layers	4 frames	
	native	4 cameras, 4,000 particles, 5 layers	261 frames	
canneal	simsmall	100,000 elements	10,000 swaps per step, 32 steps	
	simmedium	200,000 elements	15,000 swaps per step, 64 steps	
	simlarge	400,000 elements	15,000 swaps per step, 128 steps	
	native	2,500,000 elements	15,000 swaps per step, 6,000 steps	
dedup	simsmall		10 MB data	
	simmedium		31 MB data	Data affects behavior
	simlarge		184 MB data	
	native		672 MB data	
facesim	simsmall	80,598 particles, 372,126 tetrahedra	1 frame	
	simmedium	80,598 particles, 372,126 tetrahedra	1 frame	Complex scaling challenging
	simlarge	80,598 particles, 372,126 tetrahedra	1 frame	
	native	80,598 particles, 372,126 tetrahedra	100 frames	
ferret	simsmall	3,544 images, find top 10 images	16 queries	
	simmedium	13,787 images, find top 10 images	64 queries	
	simlarge	34,793 images, find top 10 images	256 queries	
	native	59,695 images, find top 50 images	3,500 queries	
fluidanimate	simsmall	35,000 particles	5 frames	
	simmedium	100,000 particles	5 frames	
	simlarge	300,000 particles	5 frames	
	native	500,000 particles	500 frames	
freqmine	simsmall	250,000 transactions, min support 220		
	simmedium	500,000 transactions, min support 410		Data affects behavior
	simlarge	990,000 transactions, min support 790		
	native	250,000 transactions, min support 11,000		
raytrace	simsmall	480×270 pixels, 1 million polygons	3 frames	
	simmedium	960×540 pixels, 1 million polygons	3 frames	Data affects behavior
	simlarge	$1,920 \times 1,080$ pixels, 1 million polygons	3 frames	
	native	$1,920 \times 1,080$ pixels, 10 million polygons	200 frames	
streamcluster	simsmall	4,096 points per block, 32 dimensions	1 block	
	simmedium	8,192 points per block, 64 dimensions	1 block	
	simlarge	16,384 points per block, 128 dimensions	1 block	
	native	200,000 points per block, 128 dimensions	5 blocks	
swaptions	simsmall		16 swaptions, 5,000 simulations	
	simmedium		32 swaptions, 10,000 simulations	
	simlarge		64 swaptions, 20,000 simulations	
	native		128 swaptions, 1,000,000 simulations	
vips	simsmall		$1,600 \times 1,200$ pixels	
	simmedium		$2,336 \times 2,336$ pixels	
	simlarge		$2,662 \times 5,500$ pixels	
	native		$18,000 \times 18,000$ pixels	
x264	simsmall	640×360 pixels	8 frames	
	simmedium	640×360 pixels	32 frames	Data affects behavior
	simlarge	640×360 pixels	128 frames	
	native	$1,920 \times 1,080$ pixels	512 frames	

Table 5.2: Overview of PARSEC inputs and how they were scaled. Cases in which the exact contents of the input data can have a strong impact on the code path or the characteristics of the program are marked.

The most difficult inputs to scale are the ones of `freqmine`. They exhibit no linear component, which means that any form of input scaling might alter the characteristics of the workload significantly. Moreover, `freqmine` parses and stores its input data internally as a frequent-pattern tree (FP-tree) that will be mined during program execution. An FP-tree is a compressed form of the transaction database that can be traversed in multiple ways. This makes the program behavior highly dependent on the exact properties of the input data, which might further amplify the problem.

The complex input components of the `facesim` inputs have not been scaled at all. Doing so would require generating a new face mesh, which is a challenging process. Significant reductions of the mesh resolution can also cause numerical instabilities. The three simulation inputs of `facesim` are therefore identical and should be considered as belonging to the `simlarge` input set.

Besides `freqmine` three more programs significantly alter their behavior depending on the data received. `Dedup` builds a database of all unique chunks that are encountered in the input stream. Less redundancy in the input stream will cause larger working sets. `Raytrace` follows the path of light rays through a scene. Small alterations of the scene or the movement of the camera might cause noticeable changes of the execution or working set sizes. However, in natural scenes with realistic camera movement this effect is likely to be small if the number of light rays is sufficiently large because their fluctuations will average out due to the law of large numbers. Finally, `x264` uses a significantly larger frame size for its `native` input. Just like `vips` the program breaks an input frame into smaller chunks of fixed size and processes them one at a time. However, `x264` must keep some frames in memory after they have been processed because it references them to encode subsequent frames. This property increases working set sizes and the amount of shared data with the frame size and is the reason why the image resolution of the input is classified as a complex input component.

5.3.2 General Scaling Artifacts

One common scaling artifact caused by linear input scaling is an exaggerated warmup effect because the startup cost has to be amortized within a shorter amount of time. Furthermore, the serial startup and shutdown phases of the program will also appear inflated in relation to the parallel phase. This is an inevitable consequence if workloads are to use

inputs with working sets comparable real-world inputs but with a significantly reduced execution time - the programs will have to initialize and write back a comparable amount of data but will do less work with it.

Consequently, all characteristics will be skewed towards the initialization and shutdown phases. In particular the maximum achievable speedup is limited due to Amdahl's Law if the whole execution of the program is taken into consideration. It is important to remember that this does not reflect real program behavior. The skew should be compensated for by either excluding the serial initialization and shutdown phases and limiting all measurements to the Region-of-Interest (ROI) of the program, which was defined to include only the representative parallel phase, or by measuring the phases of the program separately and manually weighing them correctly. It is safe to assume that the serial initialization and shutdown phases are negligible in the real inputs, which allows one to completely ignore them for experiments. Benchmark users who do not wish to correct measurements in such a way should limit themselves to the `native` input set, which is a much more realistic description of real program behavior that exhibits these scaling artifacts to a much lesser extent.

5.3.3 Scope of PARSEC Inputs

This section briefly describes what the constraints of the PARSEC simulation inputs are and under which circumstances one can expect to see additional, noticeable scaling artifacts. The most severe limitations are the amount of parallelism and the size of the working sets.

The PARSEC simulation inputs were scaled for machines with up to 64 cores and with up to tens of megabytes of cache. These limitations define the scope of these inputs, with smaller inputs having even tighter bounds. If the inputs are used beyond these restrictions noticeable scaling artifacts such as idle CPUs caused by limited parallelism must be expected. The `native` input set should be suitable for machines far exceeding these limitations.

Reducing the size of an input requires a reduction of the amount of work contained in it which typically affects the amount of parallelism in the input. Table 5.3 summarizes the number of work units in each simulation input set. This is an upper bound on the number of cores that the input can stress simultaneously.

Program	Input Set		
	simsmall	**simmedium**	**simlarge**
blackscholes	4,096	16,384	65,536
bodytrack	60	60	60
canneal	$\leq 5.0 \cdot 10^4$	$\leq 1.0 \cdot 10^5$	$\leq 2.0 \cdot 10^5$
dedup	2,841	8,108	94,130
facesim	80,598	80,598	80,598
ferret	16	64	256
fluidanimate	$\leq 3.5 \cdot 10^4$	$\leq 1.0 \cdot 10^5$	$\leq 3.0 \cdot 10^5$
freqmine	23	46	91
raytrace	1,980	8,040	32,400
streamcluster	4,096	8,192	16,384
swaptions	16	32	64
vips	475 (50)	1369 (74)	3612 (84)
x264	8	32	128

Table 5.3: Work units contained in the simulation inputs. The number of work units provided by the inputs is an upper bound on the number of threads that can work concurrently. Any other bounds on parallelism that are lower are given in parentheses.

Workloads with noticeably low amounts of work units are bodytrack, ferret, freqmine, swaptions and x264. Ferret processes image queries in parallel, which means that the number of queries in the input limits the amount of cores it can use. This can be as little as 16 for simsmall. The amount of parallelism in the simulation input sets of swaptions is comparable. The smallest work unit for the program is a single swaption, only 16 of which are contained in simsmall. X264 uses coarse-grain parallelism that assigns whole frames to individual threads. The number of possible cores the program can use is thus restricted to the number of images in the input, which is only eight in the case of simsmall. The number of work units that vips can simultaneously process is technically limited to the cumulative size of the output buffers, which can be noticeable on larger CMPs. This limitation has been removed in later versions of vips and will probably disappear in the next version of PARSEC. The amount of parallelism contained in the bodytrack inputs is limited by a vertical image pass during the image processing phase. It was not artificially introduced by scaling, real-world inputs exhibit the same limitation. It is therefore valid to use the simulation inputs on CMPs with more cores than the given limit. The upper bound introduced by input scaling is given by the number of particles, which is nearly two orders of magnitude larger. The natural limitation of parallelism dur-

ing the image processing phase should only become noticeable on CMPs with hundreds of cores because image processing takes up only a minor part of the total execution time. The bound on parallelism for `canneal` and `fluidanimate` is probabilistic and fluctuates during runtime. It is guaranteed to be lower than the one given in Table 5.3 but should always be high enough even for extremely large CMPs.

Input scaling can also have a noticeable effect on the working sets of a workload and some reduction should be expected in most cases. However, the impact is significant in the cases of 'unbounded' workloads [11], which are `canneal`, `dedup`, `ferret`, `freqmine` and `raytrace`. A workload is unbounded if it has the qualitative property that its demand for memory and thus working sets is not limited in practice. It should therefore never fully fit into a conventional cache. Any type of input that requires less than all of main memory must be considered scaled down. For example, `raytrace` moves a virtual camera through a scene which is then visualized on the screen. Scenes can reach any size, and given enough time each part of it can be displayed multiple times and thus create significant reuse. This means the entire scene forms a single, large working set which can easily reach a size of many gigabytes.

A scaled-down working set should generally not fit into a cache unless its unscaled equivalent would also fit. Unfortunately larger working sets also affect program behavior on machines with smaller caches because the miss rate for a given cache keeps growing with the working set if the cache cannot fully contain it. Unlike for parallelism, exact bounds on cache sizes are therefore not given. An impact on cache miss rates must be expected for all cache sizes. Instead this effect is accounted for by including the cache behavior for a range of cache sizes in the approximation error.

5.4 Validation of PARSEC Inputs

This section addresses the issue of accuracy of smaller input sets. It first describes the used methodology and then reports the approximation error of the input sets relative to the entire PARSEC benchmark suite.

5.4.1 Methodology

An ideal benchmark suite should consist of a diverse selection of representative real-world programs with realistic inputs. As described in Section 5.2, an optimal selection of inputs should minimize the deviation of the program behavior for a target time limit while maintaining the diversity of the entire benchmark suite. Thus, analyzing and quantifying program behavior is the fundamental challenge in measuring differences between benchmarks and their inputs.

Program behavior can be viewed as an abstract, high-dimensional feature space of potentially unlimited size. It manifests itself in a specific way such that it can be measured in the form of characteristics when the program is executed on a given architecture. This process can be thought of as taking samples from the behavior space at specific points defined by a particular architecture-characteristic pair. Given enough samples an image of the program behavior emerges.

A set of characteristics was measured for the PARSEC simulation inputs on a particular architecture. The data was then processed with principal component analysis (PCA) to automatically eliminate highly correlated data. The result is a description of the program and input behavior that is free of redundancy.

This work defines *approximation error* as the dissimilarity between different inputs for the same program, as mentioned in Section 5.2. One can measure it by computing the pairwise distances of the data points in PCA space. Approximation error is used as the basic unit to measure how accurate a scaled input is for its program. To visualize the approximation error of all benchmark inputs a dendrogram is used which shows the similarity or dissimilarity of the various inputs with respect to each other.

This methodology to analyze program characteristics is the common method for similarity analysis, but its application to analyze approximation errors is new. Measuring characteristics on an ideal architecture is frequently used to focus on program properties that are inherent to the algorithm implementation and not the architecture [9, 11, 96]. PCA and hierarchical clustering have been in use for years as an objective way to quantify similarity [28, 31, 42, 51, 73].

Program Characteristics

For the analysis of the program behavior a total of 73 characteristics were chosen that were measured for each of the 39 simulation inputs of PARSEC 2.1, yielding a total of 2,847 sample values that were considered. This study focuses on the parallel behavior of the multithreaded programs relevant for studies of CMPs. The characteristics chosen encode information about the instruction mix, working sets and sharing behavior of each program as follows:

Instruction Mix 25 characteristics that describe the breakdown of instruction types relative to the total amount of instructions executed by the program

Working Sets 8 characteristics encoding the working set sizes of the program by giving the miss rate for different cache sizes

Sharing 40 characteristics describing how many lines of the total cache are shared and how intensely the program reads or writes shared data

The working set and sharing characteristics were measured for a total of 8 different cache sizes ranging from 1 MB to 128 MB to include information about a range of possible cache architectures. This approach guarantees that unusual changes in the data reuse behavior due to varying cache sizes or input scaling are captured by the data. The range of cache sizes that were considered has been limited to realistic sizes to make sure that the results of this analysis will not be skewed towards unrealistic architectures.

Experimental Setup

To collect the characteristics of the input sets an ideal machine was simulated that can complete all instructions within one cycle using `Simics`. An ideal machine architecture was chosen because the focus of this study is on properties inherent to the program, not in characteristics of the underlying architecture. The binaries which were used are the official precompiled PARSEC 2.1 binaries that are publicly available on the PARSEC website. The compiler used to generate the precompiled binaries was `gcc 4.4.0`.

An 8-way CMP with a single cache hierarchy level that is shared between all threads was simulated. The cache is 4-way associative with 64 byte lines. The capacity of the cache was varied from 1 MB to 128 MB to obtain information about the working set

sizes with the corresponding sharing behavior. Only the Region-of-Interest (ROI) of the workloads was characterized.

Principal Component Analysis

Principal component analysis (PCA) is a mathematical method to transform a number of possibly correlated input vectors into a smaller number of uncorrelated vectors. These uncorrelated vectors are called the principal components (PC). PCA was employed in the analysis because it is considered the simplest way to reveal the variance of high-dimensional data in a low dimensional form.

To compute the principal components of the program characteristics, the data is first mean-centered and normalized so it is comparable with each other. PCA is then used to reduce the number of dimensions of the data. The resulting principal components have decreasing variance, with the first PC containing the most amount of information and the last one containing the least amount. The Kaiser's Criterion was used to eliminate PCs which do not contain any significant amount of information in an objective way. Only the top PCs with eigenvalues greater than one are kept, which means that the resulting data is guaranteed to be uncorrelated but to still contain most of the original information.

Approximation Error

All PARSEC inputs are scaled-down versions of a single real-world input, which means the more similar an inputs is to a bigger reference input for the same program the smaller is its approximation error. After the data has been cleaned up with PCA, this similarity between any two PARSEC inputs can be measured in a straightforward manner by calculating the Euclidean distance (or L_2) between them. Inputs with similar characteristics can furthermore be grouped into increasingly bigger clusters with hierarchical clustering. This method assigns each input set to an initial cluster. It then merges the two most similar clusters repeatedly until all input sets are contained in a single cluster. The resulting structure can be visualized with a dendrogram.

Inputs which merge early in the dendrogram are very similar. The later inputs merge the more dissimilar they are. Inputs for the same workload which preserve its characteristics with respect to other programs in the suite should fully merge before they join with inputs from any other benchmarks. Ideally all inputs for the same workload form their

own complete clusters early on before they merge with clusters formed by inputs of any other workloads.

Limitations

All approximation errors are expressed relative to `simlarge`, the largest simulation input. While it would be interesting to know how `simlarge` compares to `native` or even real-world inputs, this information is of limited practical value because it is infeasible to use even bigger inputs for most computer architecture studies. For the most part, benchmark users are stuck with inputs of fairly limited size no matter how big their approximation error is. Time constraints also limit the scope of this study because I believe the more detailed behavior space exploration which becomes possible with simulation is more important than a more accurate reference point which would be feasible with experiments on real machines.

5.4.2 Validation Results

The chosen characteristics of the inputs were studied with PCA. The results of this comparison are summarized by the dendrogram in Figure 5.2.

The dendrogram shows that the inputs of most workloads form their own, complete clusters before they merge with inputs of other programs. That means that the inputs preserve the characteristics of the program with respect to the rest of the suite. These workloads are `dedup`, `canneal`, `vips`, `x264`, `freqmine`, `fluidanimate`, `bodytrack`, `swaptions` and `facesim`.

The inputs for the benchmarks `blackscholes`, `raytrace`, `streamcluster` and `ferret` merge with clusters formed by inputs of other programs before they can form their own, complete cluster. This indicates that the input scaling process skewed the inputs in a way that made the region in the characteristics space that corresponds to the workload overlap with the characteristics space of a different benchmark. This is somewhat less of an issue for `ferret`, which simply overlaps with `fluidanimate` before its inputs can fully merge. However, three inputs belonging to the `simsmall` input set merge significantly later than all other inputs. These are the `simsmall` inputs for `blackscholes`, `raytrace` and `streamcluster`. This indicates that these inputs not only start to be atypical for

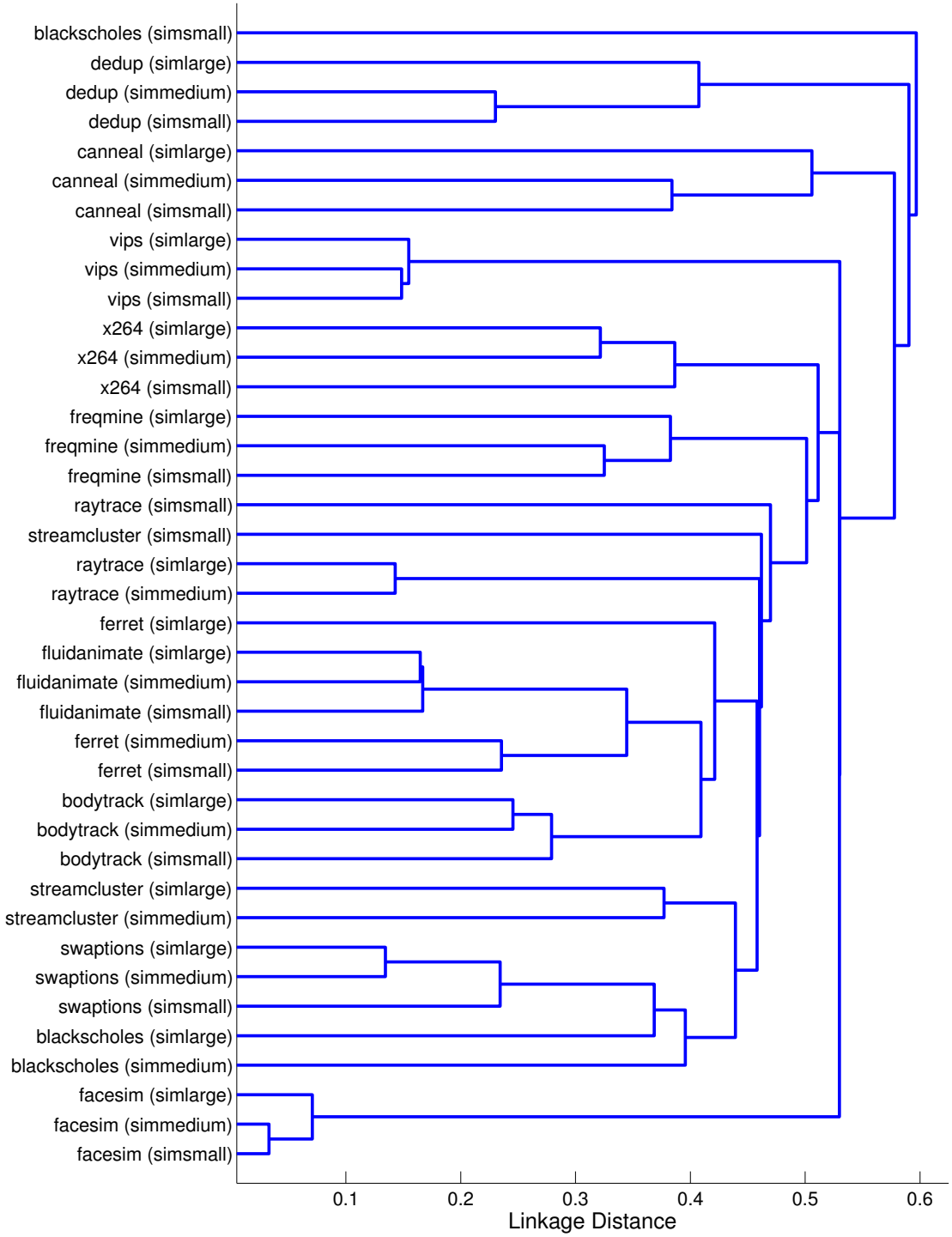

Figure 5.2: Fidelity of the PARSEC input sets. The figure shows the similarity of all inputs. The approximation error is the lack of similarity between inputs for the same workload.

their workloads, they even become atypical for the entire suite. It is important to emphasize that this increase in diversity is not desirable because it is an artificial byproduct of input scaling which does not represent real-world program behavior. In the case of `streamcluster` the working sets change significantly as its inputs are scaled, as was explained in the last section.

It is also worth mentioning which inputs merge late. While forming their own clusters first, the inputs within the clusters for `dedup`, `canneal`, `x264` and `freqmine` have a distance to each other in the dendrogram which is larger than half of the maximum distance observed between any data points for the entire suite. In these cases there is still a significant amount of difference between inputs for the same workload, even though the inputs as a whole remain characteristic for their program.

The same data is also presented in Figure 5.3. It shows directly what the distances between the three simulation inputs for each workload are without considering other inputs that might be in the same region of the PCA space. The figure shows the approximation error a_{ij} of the simulation inputs, which is simply the distance from `simlarge` in this case. As was explained earlier `simlarge` was chosen as reference point because it typically is the best input that is feasible to use for microarchitectural simulations. It is worth mentioning that the data in Figure 5.3 follows the triangle inequality - the cumulative distance from `simsmall` to `simmedium` and then from `simmedium` to `simlarge` is never less than the distance from `simsmall` to `simlarge`.

The figure shows that the inputs for `bodytrack`, `dedup`, `facesim`, `fluidanimate`, `swaptions` and `vips` have an approximation error that is below average, which means they have a high degree of fidelity. This list includes nearly all benchmarks whose inputs could be scaled with linear input scaling. The inputs with the highest fidelity are the ones for `facesim`. This is not surprising considering that the simulation inputs for that workload have not been scaled at all and are, in fact, identical. The small differences between the inputs that can be observed are caused by background activity on the simulated machine. The inputs for `blackscholes`, `canneal` and `freqmine` have a very high approximation error. These are workloads where linear input scaling has a direct effect on their working sets or which are very difficult to scale.

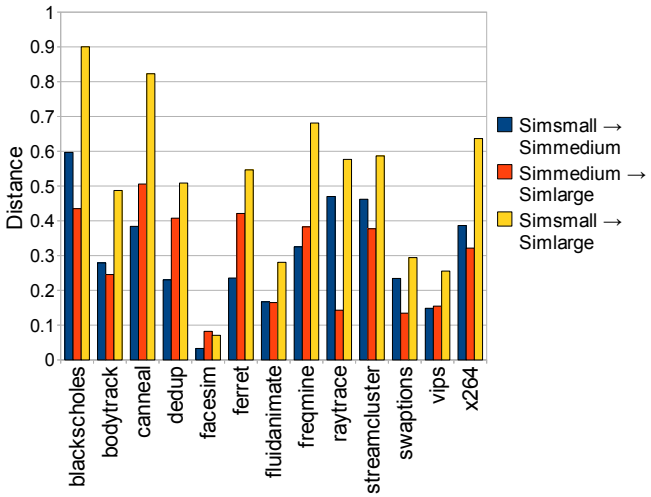

Figure 5.3: Approximation error of inputs. The approximation error or dissimilarity between any two pairs of simulation inputs for the same workload is given by the distance between the corresponding points in PCA space.

5.5 Input Set Selection

PARSEC is one of few benchmark suites which offers multiple scaled-down input versions for each workload that were derived from a single real-world input. This requires users to select appropriate inputs for their simulation study. This section discusses how to make this decision in practice.

As mentioned in Section 5.2.1, designing or selecting an input set is an optimization problem which has the structure of the classical multiple-choice knapsack problem. In practice, the simulation time budget T is often somewhat flexible. Thus, the problem can be simplified to the fractional knapsack problem, for which a simple greedy heuristic based on the benefit-cost ratio (BCR) leads to the optimal solution. With this approach first the optimal order in which to select the inputs is determined, then the simulation time budget is implicitly sized so that there is no need to take fractional inputs. The input selection problem is furthermore expressed as a selection of upgrades over `simsmall` to prevent the pathological case where the greedy heuristic chooses no input at all for a given benchmark program. The cost in this case is the increase in instructions, which is a reasonable approximation of simulation time. The task of the greedy algorithms is thus to maximize error reduction relative to the increase of instructions for each input upgrade over `simsmall`.

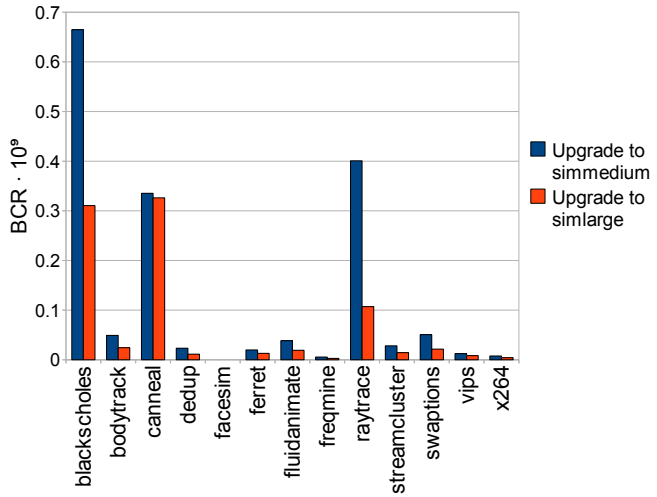

Figure 5.4: Benefit-cost ratio (BCR) of using the next larger input. The chart shows the reduction in approximation error relative to the increase of cost in billion instructions. The data for `facesim` had to be omitted due to a division by zero.

More formally, the benefit-cost ratio of upgrading from input k to input l of benchmark i is $\mathrm{BCR}_i(k,l) = -\frac{a_{il}-a_{ik}}{t_{il}-t_{ik}}$. The negative value is used because the metric is derived from the reduction of the approximation error, which is a positive benefit. The values for the variables a_{ik} are simply the distances of the respective inputs from the corresponding reference inputs of the `simlarge` input set, which is given in Figure 5.3. The values for the cost t_{ik} can be measured directly by executing the workloads with the desired inputs and counting the instructions. Figure 5.4 shows all relevant BCRs. As can be expected, the data shows diminishing returns for upgrading to a more accurate input set: Making the step from `simsmall` to `simmedium` is always more beneficial than making the step from `simmedium` to `simlarge`.

As can be seen from the data, if simulation cost is considered there are three obvious candidates which are interesting for an upgrade: Using `simmedium` instead of `simsmall` is highly attractive for the workloads `blackscholes`, `canneal` and `raytrace`. The benchmarks `blackscholes` and `canneal` are furthermore interesting for another upgrade to the `simlarge` inputs, `raytrace` to a much lesser extent. The next tier of workloads with an attractive BCR is formed by `bodytrack`, `dedup`, `ferret`, `fluidanimate`, `streamcluster` and `swaptions`. These workloads also show an increased benefit-cost ratio for an upgrade from `simsmall` to `simmedium`. An interesting fact is that it is more attractive to upgrade `blackscholes`, `canneal` and `raytrace` to `simlarge` first before

upgrading any other inputs to `simmedium`. This is primarily due to the fact that the inputs for these programs contain significantly fewer instructions than those of the other benchmarks. Benchmark users should furthermore verify that all selected inputs are used within their scope as defined in Section 5.3.3. If this is not the case it might be necessary to create custom benchmark inputs, which is discussed in the next section.

5.6 Customizing Input Sets

The input sets of the PARSEC release are scaled for a wide range of evaluation or simulation cases. However, for specific purposes one may want to consider designing their own input sets based on the methodology proposed in this chapter. This section presents a guideline for customizing inputs for more parallelism, larger working sets or higher communication intensity.

5.6.1 More Parallelism

The amount of work units available in an input can typically be directly controlled with the linear component of the input. In almost all cases, the amount of parallelism can be increased significantly so that enough concurrency is provided by the input to allow the workload to stress CMPs with hundreds of cores. However, it is important to remember that the amount of work units contained in an input are at best only potential parallelism. It is likely that not all workloads will be able to scale well to processors with hundreds of cores. Section 5.3.3 describes some technical limitations.

5.6.2 Larger Working Sets

It is straightforward to create massive working sets with PARSEC. The standard input sets already provide fairly big working sets, but as mentioned in Section 5.3, linear input scaling was used to aggressively reduce the input sizes of the simulation inputs. This can significantly reduce working set sizes if the outer-most loop has a direct impact on the amount of data that will be reused, which is especially true for benchmarks with unbounded working sets.

The general approach to obtain a larger working set with a PARSEC program is to first use complex input scaling to increase the amount of data that the program keeps in memory and then guaranteeing enough reuse of the data by linear input scaling.

It is easy to underestimate the impact of increasing working set sizes on execution time. It is known that program execution time grows at least linearly with the program's memory requirements because each item in memory has to be touched at least once. However, most programs use algorithms with complexities higher than linear, which means increasing problem sizes may lead to a prohibitively long execution time. For example, increasing the working set size of an algorithm which runs in $O(N \log N)$ time and requires $O(N)$ memory by a factor of eight will result in a 24-fold increase of execution time. If the simulation of this workload took originally two weeks, it would now take nearly a full year for a single run - more than most researchers are willing to wait.

5.6.3 Higher Communication Intensity

The most common approach to increase communication intensity is to reduce the size of the work units for threads while keeping the total amount of work constant. For example, the communication intensity of `fluidanimate` can be increased by reducing the volume of a cell because communication between threads happens at the borders between cells. However, this is not always possible in a straightforward manner because the size of a work unit might be determined by the parallel algorithm or it might be hardwired in the program so that it cannot be chosen easily.

The communication intensity is limited in nearly all lock-based parallel programs. Communication among threads requires synchronization, which is already an expensive operation by itself that can quickly become a bottleneck for the achievable speedup. Programs with high communication intensity are typically limited by a combination of synchronization overhead, lock contention or load imbalance, which means that parallel programs have to be written with the goal to reduce communication to acceptable levels. The PARSEC benchmarks are no exception to this rule.

5.7 Related Work

Closely related work falls into three categories [101]: *reduced inputs*, *truncated execution*, and *sampling*.

A popular method to reduce the number of simulated instructions of a benchmark is called *reduced inputs*. The `test` and `train` input sets of the SPEC CPU2006 benchmark suite are sometimes used as a reduced version of its authoritative `ref` input set. MinneSPEC [47] and SPEClite [91] are two alternative input sets for the SPEC CPU2000 suite that provide reduced inputs suitable for simulation. My work goes beyond previous work by proposing a framework and employing a continuous scaling method to create multiple inputs suitable for performance studies with varying degrees of fidelity and simulation cost.

Truncated execution takes a single block of instructions for simulation, typically from the beginning (or close to the beginning) of the program. This method has been shown to be inaccurate [101].

Sampling is a statistical simulation method that provides an alternative to reduced inputs. It selects small subsets of an instruction stream for detailed simulation [17, 20, 80, 97]. These sampling methods choose brief instruction sequences either randomly or based on some form of behavior analysis. This can happen either offline or during simulation via fast forwarding. Statistically sampled simulation can be efficient and accurate [101], but it requires a sophisticated mechanism built into simulators.

The importance of limiting simulation cost while preserving accuracy has motivated studies that compare sampling with reduced inputs. Haskins et al. concluded that both approaches have their uses [33]. Eeckhout et al. showed that which method was superior depended on the benchmark [50]. Finally, Yi et al. concluded that sampling should generally be preferred over reduced inputs [101]. These comparisons however did not consider that the accuracy of either method is a function of the input size. My work provides the framework to allow benchmark users to decide for themselves how much accuracy they are willing to give up for faster simulations.

Another approach is statistical simulation [52, 68, 69]. The fundamental concept of statistical simulation is to generate a new, synthetic instruction stream from a benchmark program with the same statistical properties and characteristics. These statistical properties have to be derived by first simulating a workload in sufficient detail. The statistical

image obtained that way is then fed to a random instruction trace generator which drives the statistical simulator.

5.8 Conclusions

This chapter presented a framework to scale input sets for a benchmark suite. The approach considers that benchmark inputs are approximations of real-world inputs that have varying degrees of fidelity and cost. By employing concepts of mathematical approximation theory and scientific modeling a methodology can be developed which allows benchmark creators and users to reason about the inherent accuracy and cost trade-offs in a systematic way.

The described work shows that the problem of choosing the best input size for a benchmark can be solved in an optimal way by expressing it as the multiple-choice knapsack optimization problem. The inputs can then be selected by standard algorithms so that benchmarking accuracy is maximized for a given time budget. For practical situations the problem can be further simplified so that it can be solved optimally with a simple greedy heuristic.

This chapter quantifies the approximation errors of multiple scaled input sets of the PARSEC benchmark suite and suggests a sequence of input upgrades for users to achieve higher simulation accuracies for their simulation time budgets. It also presents an analysis of the constraints of the PARSEC simulation inputs to provide users with the scope of architectures for which the inputs exhibit reasonably accurate behavior.

This chapter also gives guidelines for users to create their own input sets for PARSEC benchmark programs with a scope more fitting for their specific simulation purpose. The proposed scaling model is used to categorize the various input parameters of the PARSEC benchmarks, which gives users a better understanding of the input creation process and its impact on program behavior.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

My dissertation has proposed a methodology to develop effective benchmarks for multiprocessors. During the course of my research I used the methodology to develop the PARSEC benchmark suite which allows other researchers to use my work for their own studies. An analysis of the use of benchmark programs at top-tier computer architecture conferences shows that PARSEC has been widely accepted by other researchers who are publishing results obtained with my proposed methodology.

A direct statistical comparison of PARSEC and SPLASH-2 shows significant systematic differences between the two suites which justifies the use of new benchmarks like the ones included in PARSEC. These differences are largely driven by a shift in what constitutes a typical shared-memory multiprocessor application. Both suites focus on workloads that are typical for their time: HPC and graphics programs in the case of SPLASH-2 and desktop and server programs in the case of PARSEC, which means that systematic differences between these types of programs will also become differences between the two suites.

To gain a better understanding of the nature of modern workloads I studied the characteristics of PARSEC benchmarks in more detail. In my thesis I report details for the parallelization, working set sizes, sharing behavior and off-chip traffic of those programs. The presented data allows researchers to interpret results they have obtained with PARSEC and what the requirements of the represented workloads for future chip multiprocessors are.

For my research I also studied how to create realistic inputs for benchmark programs, a necessary step for the creation of any type of workload. I proposed to consider benchmark inputs as approximations of real program inputs because the creation of benchmark inputs usually necessitates changes that might alter the nature of the work done by the program. Using standard methods of scientific modeling and approximation theory I developed methods that can be used to quantify the approximation error and to select input approximations in a way that optimizes accuracy subject to a restricted execution time budget. These methods can be used by benchmark creators and users alike to determine the desired degree of input fidelity and cost.

The presented methodology can be reused to create new benchmark suites that give scientists a higher degree of confidence in their experimental results than existing approaches.

6.2 Future Work

Future work could build on my research by increasing the number and type of applications included in PARSEC to achieve more diversity and give scientists more choice. PARSEC focuses on desktop and server applications, future releases could also include different domains such as workloads for embedded devices, middleware or other types of software.

In a similar way the requirements of emerging applications for metrics other than performance are currently not fully understood. Especially at the frontier of computer development, in areas such as cloud or ubiquitous computing, the requirements of applications often reach the limits of technology. Examples are reliability of data warehouses or energy consumption of smart sensors.

Another area for future work is the programming model for next-generation shared-memory multiprocessors such as transactional memory. Developing parallel programs is significantly more challenging than writing serial code and a method to do so without major limitations has not yet been discovered. Most PARSEC workloads already implement multiple threading models among which benchmark users can choose. This selection allows comparative studies of the impact of the threading model on the workload, which might help to determine the impact of this choice.

Bibliography

[1] A. Alameldeen, C. Mauer, M. Xu, P. Harper, M. Martin, and D. Sorin. Evaluating Non-Deterministic Multi-Threaded Commercial Workloads. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2002.

[2] A. Alameldeen and D. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, February 2003.

[3] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63, 1991.

[4] M. P. Bailey and W. G. Kemple. The Scientific Method of Choosing Model Fidelity. In *Proceedings of the 24th Conference on Winter Simulation*, pages 791–797, New York, NY, USA, 1992. ACM.

[5] A. Balan, L. Sigal, and M. Black. A Quantitative Evaluation of Video-based 3D Person Tracking. In *IEEE Workshop on VS-PETS*, pages 349–356, 2005.

[6] P. Banerjee. *Parallel Algorithms for VLSI Computer-Aided Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[7] J. Barnes and P. Hut. A Hierarchical O(N log N) Force-Calculation Algorithm. *Nature*, 324:446–449, December 1986.

[8] L. Barroso, K. Gharachorloo, and F. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 3–14, June 1998.

[9] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In *Proceedings of the 2008 International Symposium on Workload Characterization*, September 2008.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical Report TR-811-08, Princeton University, January 2008.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[12] C. Bienia and K. Li. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[13] C. Bienia and K. Li. Characteristics of Workloads Using the Pipeline Programming Model. In *Proceedings of the 3rd Workshop on Emerging Applications and Manycore Architecture*, June 2010.

[14] C. Bienia and K. Li. Fidelity and Scaling of the PARSEC Benchmark Inputs. In *Proceedings of the 2010 International Symposium on Workload Characterization*, December 2010.

[15] F. Black and M. Scholes. The Pricing of Options and Corporate Liabilities. *Journal of Political Economy*, 81:637–659, 1973.

[16] S. Brin, J. Davis, and H. Garcia-Molina. Copy Detection Mechanisms for Digital Documents. In *Proceedings of Special Interest Group on Management of Data*, 1995.

[17] P. D. Bryan, M. C. Rosier, and T. M. Conte. Reverse State Reconstruction for Sampled Microarchitectural Simulation. *International Symmposium on Performance Analysis of Systems and Software*, pages 190–199, 2007.

[18] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *International Conference on Computer Graphics and Interactive Techniques 2004*, pages 777–786, New York, NY, USA, 2004. ACM.

[19] D. Citron. MisSPECulation: Partial and Misleading Use of SPEC CPU2000 in Computer Architecture Conferences. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 73–77, June 2003.

[20] T. Conte, M. Ann, H. Kishore, and N. Menezes. Reducing State Loss for Effective Trace Sampling of Superscalar Processors. In *Proceedings of the 1996 International Conference on Computer Design* , pages 468–477, 1996.

[21] G. Contreras and M. Martonosi. Characterizing and Improving the Performance of the Intel Threading Building Blocks Runtime System. In *International Symposium on Workload Characterization*, September 2008.

[22] M. Desbrun and M.-P. Gascuel. Smoothed Particles: A New Paradigm for Animating Highly Deformable Bodies. In *Proceedings of the 6th Eurographics Workshop on Computer Animation and Simulation*, pages 61–76, August 1996.

[23] J. Deutscher and I. Reid. Articulated Body Motion Capture by Stochastic Search. *International Journal of Computer Vision*, 61(2):185–205, February 2005.

[24] P. Dubey. Recognition, Mining and Synthesis Moves Computers to the Era of Tera. *Technology@Intel Magazine*, February 2005.

[25] K. Dudzinski and S. Walukiewicz. Exact Methods for the Knapsack Problem and its Generalizations. *European Journal of Operations Research*, 28(1):3–21, 1987.

[26] G. Dunteman. *Principal Component Analysis*. Sage Publications, 1989.

[27] Elephants Dream. Available at `http://www.elephantsdream.org/`, 2006.

[28] R. Giladi and N. Ahituv. SPEC as a Performance Evaluation Measure. *Computer*, 28(8):33–42, 1995.

[29] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[30] G. Grahne and J. Zhu. Efficiently Using Prefix-Trees in Mining Frequent Itemsets. In *Proceedings of the Workshop on Frequent Itemset Mining Implementations*, November 2003.

[31] H. Vandierendonck and K. De Bosschere. Many Benchmarks Stress the Same Bottlenecks. In *Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 57–64, 2 2004.

[32] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In W. Chen, J. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM Press, 05 2000.

[33] J. W. Haskins, K. Skadron, A. J. Kleinosowski, and D. J. Lilja. Techniques for Accurate, Accelerated Processor Simulation: Analysis of Reduced Inputs and Sampling. Technical report, University of Virginia, Charlottesville, VA, USA, 2002.

[34] D. Heath, R. Jarrow, and A. Morton. Bond Pricing and the Term Structure of Interest Rates: A New Methodology for Contingent Claims Valuation. *Econometrica*, 60(1):77–105, January 1992.

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

[36] L. Hernquist and N. Katz. TreeSPH - A Unification of SPH with the Hierarchical Tree Method. *The Astrophysical Journal Supplement Series*, 70:419, 1989.

[37] W. D. Hillis and G. L. Steele. Data-Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[38] C. J. Hughes, R. Grzeszczuk, E. Sifakis, D. Kim, S. Kumar, A. P. Selle, J. Chhugani, M. Holliman, and Y.-K. Chen. Physical Simulation for Animation and Visual Effects: Parallelization and Characterization for Chip Multiprocessors. *SIGARCH Computer Architecture News*, 35(2):220–231, 2007.

[39] J. C. Hull. *Options, Futures, and Other Derivatives*. Prentice Hall, 2005.

[40] Intel. Threading Building Blocks. Available at `http://www.threadingbuildingblocks.org/`, 2008.

[41] A. Jaleel, M. Mattina, and B. Jacob. Last-Level Cache (LLC) Performance of Data-Mining Workloads on a CMP - A Case Study of Parallel Bioinformatics Workloads. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, February 2006.

[42] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring Benchmark Similarity Using Inherent Program Characteristics. *IEEE Transactions on Computers*, 28(8):33–42, 1995.

[43] K. Hoste and L. Eeckhout. Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics. In *Proceedings of the IEEE International Symposium on Workload Characterization 2006*, pages 83–92, 2006.

[44] J. T. Kajiya. The Rendering Equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, pages 143–150, New York, NY, USA, 1986. ACM.

[45] R. M. Karp and M. O. Rabin. Efficient Randomized Pattern-Matching Algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[46] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media Processing with Streams. *IEEE Micro*, 21(2):35–46, 2001.

[47] A. J. Kleinosowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters*, 1, 2002.

[48] D. J. Kuck. A Survey of Parallel Machine Organization and Programming. *ACM Computing Surveys*, 9(1):29–59, 1977.

[49] M. Kudlur and S. Mahlke. Orchestrating the Execution of Stream Programs on Multicore Platforms. *SIGPLAN Notices*, 43(6):114–124, 2008.

[50] L. Eeckhout and A. Georges and K. De Bosschere. Selecting a Reduced but Representative Workload. In *Middleware Benchmarking: Approaches, Results, Experiences. OOSPLA workshop*, 2003.

[51] L. Eeckhout and H. Vandierendonck and K. De Bosschere. Quantifying the Impact of Input Data Sets on Program Behavior and its Applications. *Journal of Instruction-Level Parallelism*, 5:1–33, 2003.

[52] L. Eeckhout and S. Nussbaum and J. E. Smith and K. De Bosschere. Statistical Simulation: Adding Efficiency to the Computer Designer's Toolbox. *IEEE Micro*, 23:26–38, 2003.

[53] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[54] K. Li and J. F. Naughton. Multiprocessor Main Memory Transaction Processing. In *Proceedings of the First International Symposium on Databases in Parallel and Sistributed Systems*, pages 177–187, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[55] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The ALPBench Benchmark Suite for Complex Multimedia Applications. In *Proceedings of the IEEE International Symposium on Workload Characterization 2005*, October 2005.

[56] S.-W. Liao, Z. Du, G. Wu, and G.-Y. Lueh. Data and Computation Transformations for Brook Streaming Applications on Multiprocessors. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 196–207, Washington, DC, USA, 2006. IEEE Computer Society.

[57] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. WebDocs: A Real-Life Huge Transactional Dataset. In *2nd IEEE ICDM Workshop on Frequent Itemset Mining Implementations 2004*, November 2004.

[58] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200. ACM Press, June 2005.

[59] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Ferret: A Toolkit for Content-Based Similarity Search of Feature-Rich Data. In *Proceedings of the 2006 EuroSys Conference*, pages 317–330, 2006.

[60] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 950–961, 2007.

[61] U. Manber. Finding Similar Files in a Large File System. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Fransisco, CA, USA, October 1994.

[62] K. Martinez and J. Cupitt. VIPS - A Highly Tuned Image Processing Software Architecture. In *Proceedings of the 2005 International Conference on Image Processing*, volume 2, pages 574–577, September 2005.

[63] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. In *ACM Transactions on Modeling and Computer Simulation*, volume 8, pages 3–30, January 1998.

[64] M. Müller, D. Charypar, and M. Gross. Particle-Based Fluid Simulation for Interactive Applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[65] R. Narayanan, B. Özisikyilmaz, J. Zambreno, G. Memik, and A. N. Choudhary. MineBench: A Benchmark Suite for Data Mining Workloads. In *Proceedings*

of the IEEE International Symposium on Workload Characterization 2006, pages 182–188, 2006.

[66] R. M. Nauss. The 0-1 Knapsack Problem with Multiple-Choice Constraints. *European Journal of Operations Research*, 2(2):125–131, 1978.

[67] R. Nock and F. Nielsen. Statistical Region Merging. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:1452–1458, 2004.

[68] 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*, pages 15–24, Washington, DC, USA, 2001. IEEE Computer Society.

[69] S. Nussbaum and J. E. Smith. Statistical Simulation of Symmetric Multiprocessor Systems. In *Proceedings of the 35th Annual Simulation Symposium*, page 89, Washington, DC, USA, 2002. IEEE Computer Society.

[70] L. O'Callaghan, A. Meyerson, R. M. N. Mishra, and S. Guha. High-Performance Clustering of Streams and Large Data Sets. In *Proceedings of the 18th International Conference on Data Engineering*, February 2002.

[71] OpenMP Architecture Review Board. OpenMP Application Program Interface. Available at `http://www.openmp.org/`, 2008.

[72] G. Ottoni, R. Rangan, A. Stoler, and D. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the 38th Annual International Symposium on Microarchitecture*, page 12, 2005.

[73] A. Phansalkar, A. Joshi, and L. K. John. Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 412–423, New York, NY, USA, 2007. ACM.

[74] D. Pisinger. A Minimal Algorithm for the Multiple-Choice Knapsack Problem. *European Journal of Operational Research*, 83:394–410, 1994.

[75] D. Pnueli and C. Gutfinger. *Fluid Mechanics*. Cambridge University Press, 1992.

[76] S. Quinlan and S. D. Venti. A New Approach to Archival Storage. In *Proceedings of the USENIX Conference on File And Storage Technologies*, January 2002.

[77] M. Rabin. Fingerprinting by Random Polynomials. Technical Report TR-15-81, Harvard University, 1981.

[78] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August. Decoupled Software Pipelining with the Synchronization Array. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 177–188, 2004.

[79] Y. Rubner, C. Tomasi, and L. J. Guibas. The Earth Mover's Distance as a Metric for Image Retrieval. *International Journal of Computer Vision*, 40:99–121, 2000.

[80] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large-Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, New York, NY, USA, 2002. ACM.

[81] E. Sifakis, I. Neverov, and R. Fedkiw. Automatic Determination of Facial Muscle Activations from Sparse Motion Capture Marker Data. *ACM Transactions on Graphics*, 24(3):417–425, 2005.

[82] P. Sinha and A. A. Zoltners. The Multiple-Choice Knapsack Problem. *Operations Research*, 27(3):503–515, 1979.

[83] SPEC CPU2006. Available at `http://www.spec.org/cpu2006/`, 2006.

[84] SPEC OMP2001. Available at `http://www.spec.org/cpu2001/`, 2001.

[85] N. T. Spring and D. Wetherall. A Protocol-Independent Technique for Eliminating Redundant Network Traffic. In *Proceedings of ACM SIGCOMM*, August 2000.

[86] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205. ACM, June 1994.

[87] J. Subhlok and G. Vondran. Optimal Latency-Throughput Tradeoffs for Data Parallel Pipelines. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 62–71, New York, NY, USA, 1996. ACM.

[88] J. Teran, E. Sifakis, G. Irving, and R. Fedkiw. Robust Quasistatic Finite Elements and Flesh Simulation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 181–190, New York, NY, USA, 2005. ACM Press.

[89] The Open Group and IEEE. IEEE Std 1003.1, 2004.

[90] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[91] R. Todi. SPEClite: Using Representative Samples to Reduce SPEC CPU2000 Workload. In *Proceedings of the 2001 IEEE International Workshop of Workload Characterization*, pages 15–23, Washington, DC, USA, 2001. IEEE Computer Society.

[92] D. Vatolin, D. Kulikov, and A. Parshin. MPEG-4 AVC/H.264 Video Codecs Comparison. Available at `http://compression.ru/video/codec_comparison/pdf/msu_mpeg_4_avc_h264_codec_comparison_2007_eng.pdf`, 2007.

[93] L. Verlet. Computer Experiments on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review*, 159:98–103, 1967.

[94] T. Whitted. An Improved Illumination Model for Shaded Display. *Commun. ACM*, 23(6):343–349, 1980.

[95] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC Video Coding Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.

[96] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of*

the 22nd International Symposium on Computer Architecture, pages 24–36, June 1995.

[97] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 84–97, 2003.

[98] G. Xu. A New Parallel N-Body Gravity Solver: TPM. *The Astrophysical Journal Supplement Series*, 98:355, 1995.

[99] T. Y. Yeh, P. Faloutsos, S. Patel, and G. Reinman. ParallAX: An Architecture for Real-Time Physics. In *Proceedings of the 34th International Symposium on Computer Architecture*, June 2007.

[100] J. Yi, R. Sendag, L. Eeckhout, A. Joshi, D. Lilja, and L. K. John. Evaluating Benchmark Subsetting Approaches. In *Proceedings of the International Symposium on Workload Characterization*, pages 93–104, October 2006.

[101] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and Comparing Prevailing Simulation Techniques. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 266–277, Washington, DC, USA, 2005. IEEE Computer Society.