Comparing Doom 3, WarCraft III, PBRT, and MESA Using Micro-architecturally Independent Characteristics

Jiayuan Meng, Henry Cook, Kevin Skadron, Dee A.B. Weikle
Department of Computer Science
University of Virginia
{jm6dg, hmc3z, skadron, dweikle}@cs.virginia.edu
Technical Report Number: CS-2007-04

February 10, 2007

Abstract

Computer games have become a driving application in the personal computer industry. For computer architects designing general purpose microprocessors, understanding the characteristics of this application domain is important to meet the needs of this growing market demographic. In addition, games and 3D-graphics applications are some of the most demanding personal computer programs. Understanding the characteristics of these applications is complicated, though, by their competitive market. Source code is generally unavailable and benchmarks are geared more toward performance in the gaming world than understanding program characteristics important in architecture. To facilitate our architecture research, we have performed a characterization of two popular games, Doom 3 and Warcraft III, and compared them to two publicly available programs, PBRT (Physically Based Ray Tracer) and MESA, one of the Spec2000 benchmarks for 3D graphics, using microarchitecturally-independent metrics. The dynamic execution of the programs was analyzed with modified Macintosh development tools and a comparison made with principal component analysis techniques. We found that the characteristics of these games differ from each other and the publicly available programs, particularly in memory reference patterns, and in their use of specialized instructions. From this preliminary investigation, we determine that games have unique characteristics, but that PBRT is more similar to both than MESA.

1 Introduction

Understanding the variety and individual characteristics of application domains is critical to designing high performance architectures for not only the scientific community, but for the personal computer market as well. In the personal computer market, this task is complicated by the business goals of the companies who write popular software. To remain competitive, these companies must keep private the source code for their applications, along with any techniques they use to make their programs faster, more useful, or (as is often the case for games) more exciting and realistic than their competition. This secrecy makes it very difficult for researchers in computer architecture, especially those in academia, to understand the low-level operation of these applications well enough to design general-purpose architectures to support them.

With current trends toward multi-core architectures, the research community has a rare opportunity to shape the essential foundation of the next era of general purpose architectures. This opportunity makes it essential to understand the general characteristics of applications, because with multi-core architectures the number of possible design solutions increases exponentially making it impossible to run detailed simulations and experiments that span the design space. By characterizing applications, the community as a whole has access to similarities that can be used to group similar applications and run fewer simulations to predict performance. Micro-architectural metrics enable new visions of general-purpose designs, by illuminating characteristics of the application, regardless of the platform on which it is executed. In addition, characterizations of applications can be used to identify new, unique benchmarks, and combine them in the most efficient ways possible for architecture experiments and analysis.

Computer games are a huge market segment and design driver in the personal computer industry. They are computationally intensive, often including artificial intelligence, physical simulation, realistic graphics, and audio components operating in parallel. In addition, their interactive nature introduces real-time latency constraints. As a result, they stress personal computers more than almost any other consumer workload. Games today are CPU bound, not GPU bound. They are also one of the few workloads likely to scale to multiple cores. Yet, detailed evaluation of these games for the purpose of architecture design is largely unexplored in academia. In this paper we perform a preliminary characterization of two popular games, Doom 3 and Warcraft III, and compare them to two publicly available programs, PBRT (Physically Based Ray Tracer) and MESA, one of the SPEC2000 benchmarks. Our goal is to find publically available programs that we could include in a benchmark suite to represent one or more games. We compare them using the microarchitecturally independent characteristics described in Eeckhout, Sampson, and Calder [3]. Each application is run on a PowerPC G4 processor for as long as possible to obtain basic block file descriptions as described in Sherwood, Perelman, and Calder [11]. Modified Macintosh development tools [4] and the Turandot [7] simulator are used to complete the analysis on the best simulation point as determined by Simpoint [8]. The results are then compared to determine what, if any, similarities exist between these applications.

The contributions of this paper are:

1) We perform a characterization of sections of Doom 3, Warcraft III, PBRT, and MESA to determine architecturally independent similarities.

2) We use a method that does not rely on source code to perform the characterization, and make the Macintosh code for this analysis generally available.

3) We show preliminary results of a principal component analysis comparing our data to the data obtained by Eeckhout, Sampson, and Calder [3] on the SPEC2000 benchmark suite using the Alpha instruction set architecture (ISA) where we use the Power ISA. The differences between the versions of mesa indicate that such a comparison may not be completely appropriate, motivating a need for better methods of comparison for applications across ISAs.

The rest of this paper is organized as follows: Section 2 explains related work, Section 3 gives background information on the different applications being analyzed, Section 4 describes the methodology used in depth, Section 5 shows results of the analysis, Section 6 enumerates specific conclusions, and Section 7 outlines plans for future work.

2 Related Work

Eeckhout, Sampson, and Calder [3], use the microarchitecturally-independent characteristics outlined in Phansalkaal. [9] to analyze the SPEC2000 benchmark suite and describe a method of reducing simulations with this suite to a subset of the most unique sections to obtain the most accurate results possible with the least simulation time. They emphasize the importance of performing microarchitecturally-independent characterizations to enable applications to drive the architecture and for unique characteristics of applications to drive their inclusion in benchmark suites. Phansalkar et al. [9] uses these characteristics to analyze the similarities of the SPEC benchmark suites over time and determine subsets of the SPEC suites to run. They find that there exists a great deal of similarity across all of the SPEC suites with the SPEC2000 suite slightly more varied than previous suites. The main difference over time in these benchmarks is the increase in the dynamic instruction count and the increase in poor temporal data locality. Both Eeckhout and Phansalkar use principal component analysis as we do here to determine similarity with a wide-variety of parameters. They use k-means clustering to determine similar clusters of benchmarks, while we use a distance measure on the principal components to compute distance from our target applications. The only other work using microarchitecturally-independent metrics we are aware of is the work by Driesen et al. [**?**] where they characterize several small familar programs such as fibonacci, nqueens, tower, quicksort, etc. using two metrics, footprints and unified prediction profiles for the memory system. They do show instruction mixes, but there ends the similarity.

Due to the closed-source nature of games, the architecture research community has not performed many characterizations of these programs. Two papers, Bangun, Dutkiewicz, and Anido [**?**] and and Feng, Chang, Feng, and Walpole [**?**] characterize the network traffic of popular on-line multi-player games. Feng et al. emphasize the importance of the class of games known as "first-person shooters" and include Doom in their list of such games. They demonstrate that the traffic consists of large, highly periodic, bursts of small packets and it targets the saturation of the narrowest, last-mile link to enable the slowest player to stay in the game. Bangun et al. focusses on the distribution of the traffic for different games, noting that the distributions are dependent on the game, but in some cases independent of the number of players. Mitra

and Chiueh [**?**] perform a characterization of 3D graphics workloads with the architectural implications for graphics processors. They use three sets of benchmarks, including Viewperf OpenGL performance evaluation benchmarks, a time-demo of QuakeII, and three VRML 1.0 models. The characteristics that they evaluate have little relationship to those we use and consist of graphics specific metrics including single-frame geometry bandwidth, total primitives, number of vertex/primitive, resolution requirements, and rasterization requirements. None of the above provide the kind of detailed general-purpose analysis we perform here.

3 Application Descriptions

When considering which applications to include in our study, we identified games that are representative of the popular trends in structure, design and graphics. To this end, we chose two bestselling games from the past three years available on the Macintosh architecture: Doom 3 and Warcraft III: Reign of Chaos. Doom 3 is a science-fiction/horror first person shooter (FPS) game. Warcraft III is a fantasy real-time strategy (RTS) game. Both applications are are several years old, but they both feature graphics and game play which were cutting edge at the time of their release and which still create challenging workloads for some modern hardware. These two games present a representative slice of the demands placed on the workstations of gaming users.

PBRT is an open source physically-based ray tracing program published by Pharr et al. and Humphreys et al. in 2004 [10]. It is able to produce photo-realistic rendering results offline rather than in realtime. Nevertheless, it provides an open reference in the context of physically-based rendering, which is one of the main interest in modern games. MESA is a free OpenGL work-alike library. Since it supports a generic frame buffer, it can be configured to have no operating system or windowing system dependencies. Any number of client programs can be written to stress floating point, scalar or memory performance (or a mix).

There are no good open-source game benchmarks. The goal in analyzing these open-source programs is to investigate whether they have similar characteristics to the selected game applications. Without access to source code, it is difficult to analyze how much computational effort the games put into rendering and related graphics tasks, as compared to tasks associated with animation, simulation or artificial intelligence. By comparing the most representative phase of these games execution with those of the open-source rendering programs, we attempt to determine whether the rendering programs can serve as valid proxies for their commercial counterparts. Uncovered architecturally independent similarities will allow us to have greater confidence in open-source benchmarks as representative of modern industry trends, while any differences will provide insight into how we can better design experiments characterizing game programs. More details are provided on each application in the subsections below.

3.1 Doom 3

As mentioned above, Doom 3 is a science-fiction/horror first person shooter which was developed by id Software and published by Activision in August 2004; the Macintosh version was released in March 2005. (See http://www.doom3.com/) Like most modern games, Doom 3 uses multiple threads to accomplish the different tasks involved in managing and presenting the game. These tasks include artificial intelligence strategies both to assist and oppose the player, receiving and processing user input, score and record keeping calculations, and realistic image creation and rendering.

Doom 3 is famous for not only its custom-built graphics engine, but also for the way this engine is employed to create surprisingly immersive and life-like environments and characters [5]. Games from the first person shooter genre generally emphasize fast-paced action, requiring quick reflexes and high levels of hand-eye coordination on the part of the user, and Doom 3 is no exception. For this reason, popular benchmarks frequently use statistics such as "frames per second' to measure a game's performance on a particular hardware system [1]. For the user to remain satisfied with the virtual experience provided, a minimal apparent speed of execution and interactivity must be maintained. However, statistics like frames per second are more indicative of the capabilities of a specific hardware configuration than they are helpful to us in understanding the underlying characteristics of a given application.

Due to our machines' capabilities, and to place maximal load on the processor, rather than the graphics card, the program is run with the graphical quality set to "Low" but with all image quality features enabled. These settings were also the ones recommended for the specific hardware, and represent the setup which would likely be chosen by a theoretical user. Generally the effects of a "Low" quality graphics setting consist of compressed mapping and textures [1].

3.2 Warcraft III

Warcraft III is a fantasy real-time strategy (RTS) game produced and developed by Blizzard Entertainment which

was released in July 2002. Real-time strategy games place the user in control of entire armies or cities over course of extended tactical encounters and conflicts. Warcraft III is also a multi-threaded application, and must perform tasks similar to those outlined for Doom 3 above. Due the scale and perspective of its gameplay, Warcraft III places less emphasis on immersive or realistic graphics, but the sheer number of units needing to be rendered and managed still can create heavy workloads. For this reason, frames per second is still the favored performance measurment among industry benchmarkers ([2], for example). We choose to run Warcraft with the default graphical settings for our hardware configuration.

While Doom 3's core gameplay is based on forcing the user to make frequent high-speed decisions, in the Warcraft gameplay paradigm the user is primarily in charge of macromanagement, or the long-term planning and tactical aspects of the game. The individual actions of units controlled by both players and computer opponents are all directed on a second-to-second scale by the game's artificial intelligence functionality. In general, this creates an environment where user input affects the flow of application execution only over longer time scales, but consistent and rapid feedback rates are still required for user satisfaction.

3.3 PBRT

Ray tracing is a long-standing fundamental technique in computer graphics. This technique works to render a 3D scene by casting rays of light from the camera, tracing their paths (which might be reflected or bent by intervening objects and surfaces), and computing the final color of each ray. We chose PBRT (Physically Based Ray Tracer) [10] as a representative ray tracing program due to its open source accessibility, broad coverage of ray tracing techniques, and well-designed infrastructure. PBRT is a photo-realistic rendering program written by Pharr and Humphreys for their book, Physically-Based Rendering [10]. The techniques included in PBRT include camera simulation, ray-object intersections, light distributions, visibility, surface scattering, recursive ray tracing, and ray propagation. Sampling theory and the Monte Carlo integration method are two significant components of its operation.

PBRT's main rendering loop contains four main components: the sampler, the camera, the integrators and the film. It takes as input a scene file, which contains information for the 3D model (mainly triangle lists), light source information, camera position, etc. The sampler then creates a set of sample points on the film plane. Stratified sampling is often used to generate the sampling patterns. For each sample point, the camera generates a ray, which travels from the film through the lens to the object world. The ray then passes through several physically modeled processes. It is tested for intersection with 3D objects. On an intersection, one or multiple new rays are generated for reflection and refraction. When the ray passes through translucent volumes such as jade or smode, scattering will also take place.

Radiance is calculated by the integrators according to the light source, object material, and other physical constraints during any intersections or along the ray's path. Occlusion tests are also made in order to generate shadows. Integration is the key operation in rendering tasks such as volume scattering and soft shadows from area light. Monte Carlo integration is a method for using random sampling to estimate the values of the integrals, which generally do not have analytic solutions.

Finally, the integrators send the sample ray and its associated radiance to the film, which stores the contribution for that ray in the image. When all the samples are processed, the final image is generated. The output image is in OpenEXR format, which preserves the image's high dynamic range information. The format developers' web site, www.openexr.org, has full details on the image file format.

3.4 Mesa

MESA is a free OpenGL work-alike 3-D graphics library authored by Brian E. Paul and written in ANSI C. Since it supports a generic frame buffer it can be configured to have no operating system or windowing system dependencies. Any number of client programs can bewritten to stress floating point, scalar or memory performance (or a mix thereof). Output can be written to image files for verification. The input data is a 2D scalar field. The scalar data is mapped to height, creating a 3D object with explicit vertex normals. Contour lines are mapped onto the object as a 1D texture. The output is a 2D image file in PNG format. More information can be found at http://www.mesa3d.org/.

4 Methodology

All applications are run on an Apple Macintosh Mini with a 1.5 GHz PowerPC G4 7447A processor and a Radeon9200 graphics card. We use a modified version of the Macintosh tracing tool Amber, called AmberBBV, to trace as many instructions as possible in each application while collecting basic block vectors (bbv) as de-

scribed in Sherwood, Perelman and, Calder [11]. The initial block size used is 100M. This detailed basic block information is then translated to a BBV file for 600M intervals. Since our comparison and characterization involve games, which can run forever, it is not feasible to analyze every interval and use clustering analysis as described in [3]. Therefore, we analyze only the most representative 600M interval in the trace with 30 billion instructions - the longest trace we are able to capture. SimPoint [8] is used for phase analysis to determine the starting simulation point for the highest weighted 600M interval. In the final tracing step Amber is used again to trace this interval. We use the Macintosh tool acid and Turandot(a PowerPC architecture simulator [7]) to capture all of the microarchitecturally-independent characteristics described below. Instruction mix, working set and data stream strides can be retrieved from acid, which produces memory access information at both instruction level and program level. A modified version of Turandot is used to capture information about register traffic, branch predictability and instruction-level parallelism.

Several aspects of this process are explained in more detail in the subsections below. Section 4.1 outlines the specific characteristics identified and how our method may differ from that of Eeckhout, Sampson, and Calder [3]. Section 4.2 describes the issues and methods behind capturing traces from the interactive game applications. Section 4.3 details the changes made to the Macintosh development tools and the Turandot simulator to capture the microarchitecturally-independent characteristics used for our evaluation. Finally, section 4.4 gives an overview of the statistical analysis methods used to analyze the gathered data.

4.1 Characteristics

The microarchitecturally-independent characteristics used in our evaluation consist of instruction mix, instruction-level parallelism, register traffic, working set size, data stream strides, and branch-predictability. We attempt to match the measures used by Eeckhout, Sampson, and Calder [3] and Phansalkar, Joshi, Eeckhout, and John [9]as closely as possible. A brief description of these characteristics is included here, but for details please refer to the original papers.

Instruction mix. Our focus is on games and rendering programs where there are often a significant number of vector operations. Therefore, we slightly modified Eeckhout et al.'s approach and include the percentage of loads, stores, branches, integer operations, floating-point operations, cache control instructions, and vector operations.

Instruction-level parallelism (ILP). To quantify the amount of ILP, we consider an idealized out-of-order processor model where only the window size is limited. We compute the number of independent instructions there are within the current window for window sizes of 32, 64, 128 and 256 in-flight instructions. No-ops are not counted as independent instructions. Moreover, when measuring dependence due to memory access, we assume that reads and writes are performed in blocks of 32 bytes.

Register traffic. The register characteristics collected include the average number of input operands to an instruction, the average degree of use, and register dependency distance statistics. The average degree of use is defined as the average number of times a register instance is read since it is written. The register dependency distance is the number of dynamic instructions between writing a register and reading it.

Working set. The working set size is computed separately for the instruction and data streams. The number of unique 32-byte blocks touched and the number of unique 4KB pages touched for both instruction and data accesses are recorded for each execution interval (600M instructions).

Data stream strides. Data stream strides are either considered local or global. Local strides are determined by capturing the distribution of strides exhibited by each individual load or store in the program, i.e. the distances between the successive memory accesses of that instruction. Frequency counts for are maintained for certain stride distances (i.e. the count a local load is < 8, < 64, < 512 etc.) Global strides are similar to local strides, but instead of keeping track of the stride of each load or store in the program, we keep track of the stride between temporally adjacent memory accesses (loads and stores are tracked separately). These vectors can become very large so memory vectors are not allowed to grow beyond 200,000 elements. All indices are then calculated modulo the max vector size. [6]

Branch predictability The most important characteristic of branch predictability is the dynamic predictor performance for the program during execution. To capture branch predictability in a micro-architecturally independent fashion we use the same PPM predictors as Eeckhout et al. [3]. Eeckhout et al. consider four variations of the PPM predictor: GAg, PAg, GAs and PAs. G means global branch history, whereas P stands for per-address or local branch history; g means one global predictor table shared by all branches, and s means separate tables per branch. They emphasize the view that the PPM predictor is a theoretical basis for branch-prediction because it attains the upper-limit performance of current branch predictors. In

our experiment, we set the order of the PPM predictor to 13, which is sufficiently large to satisfy the upper bound requirement.

4.2 Interactive Complications

One of the major challenges to performing the kind of phase analysis provided by SimPoint is capturing a representative fraction of the application's execution phases. When there is no set run time for a program because the period of execution is dependent on the whims of a user, the selection of starting and ending trace points could become arbitrary choices within an infinite stream of instructions and unknown number of phases. To capture a representative area of execution, we use the 'Quick Load' feature of the game applications to script the loading of an interesting game state and used further scripting to ensure that tracing occurred a measurable distance after the loading process began. For Doom 3, 30 billion instructions proves to be about the longest possible trace lengths from a purely practical point of view, and 500 million instructions are initially skipped in attempt to bypass the initial loading state.

Another challenge specific to the user interactive applications domain is the dichotomy between ensuring experimental repeatability and providing reasonable input to the application so as to generate realistic behavior. This contrasts with the deterministic nature of tasks such as rendering (i.e. PBRT and MESA). In general, scripting user input is a valid solution to this problem, but in the case of our very real-time behavior dependent applications (games) we encounter problems. For basic in-application control and input we use AppleScript, but due to the massive slowdown experienced by traced applications, it is impossible to provide timed inputs to match the input capabilities of the traced application and still produce 'realistic behavior' (i.e. spatial navigation through a Doom 3 level). In general, we compensate for this by simply choosing to trace a game state not dependent solely on user input for interesting behavior. If graphical demands, artificial intelligence, and automatic services are active enough at certain point in game play, the effect of a brief lack of user input on execution is negligible. To accomplish this, the game is navigated manually to an interesting state (the conclusion of a cinematic with an enemy attack in the case of Doom 3), and then the 'quick save' and 'quick load' features are used to allow tracing to begin with the game in same the precise state every time. Thus we preserve repeatability while still capturing an interesting and realistic slice of program execution.

Our approach to extracting traces Warcraft's execution is nearly identical to the method used with Doom 3 as outlined above. The actual tracing parameters are precisely the same, and only minor modifications are needed to the script in charge of setting up the application state and providing user input. As discussed in 3.2, one benefit of Warcraft III's game play paradigm over that of Doom 3 in this situation is that the user is mainly in charge of macromanagement. Individual units' actions are all mostly controlled by the game's artificial intelligence algorithms. For this reason, given a complex game state examined over a scale of several seconds, there is no reason to require any user input at all (this behavior may even be considered typical). By customizing a multi-player scenario it is possible to create a game state as complex as may be desired. However, in the specific trace analyzed in this paper, we choose to use a default scenario provided by the manufacturers for scripting simplicity; we deem it complicated enough to meet our phase coverage requirements. As with Doom 3, we automate the process of opening the application and loading the desired state, whereupon tracing automatically begins. 30 billion instructions are traced. Again, as with Doom 3, a fixed window of 500 million instructions corresponding to the loading phase are skipped. The resulting basic block vector is processed in the same way, resulting a detailed trace of the block associated with the execution phase weighted highest by our SimPoint analysis.

4.3 Analyzing Software

AmberBBV is an extension based on the external library of Amber, a performance analyzing tool for the PowerPC architecture designed to trace program instructions. The external library provides an interface for analyzing the trace. The trace data is then parsed and fed to the BBV module which produces the BBV profile. To identify a basic block, the programs simply check each parsed instruction to see whether it is a branch instruction. If so, then we have reached the end of a basic block and must begin a new basic block. The address of the first instruction in a basic block is used as the identifier for that block. We keep all the basic block records in a hash table and increment the counter of a specific basic block when an instruction in that basic block is executed. Our BBV module is based on BBTracker, which was released by Calder et al. for producing BBV data with SimpleScalar [11]. The BBV file can then be used for phase analysis and to produce weighted simulation points. The extended features of AmberBBV include:

1. Parsing the trace on the fly to generate BBV records. AmberBBV analyzes the trace data to produce BBV

records. Since traces are usually huge in size, and sometimes all we need is the BBV record file, users can decide whether or not to save the trace while producing the BBV file. A BBV file is produced for each thread.

2. Configuring the trace options for different threads. When examining the traces from games, the significant phases of execution may occur at different times for different threads. However, Amber doesn't distinguish configurations between threads, hence it is hard to monitor the trace progress of a particular thread. To provide more sophisticated per thread analysis, AmberBBV keeps a separate record for each thread, and users can specify the number of skipped instructions for each thread explicitly.

Acid is a Macintosh development tool that analyzes the trace files [4]. From an instruction trace it can record and calculate the instruction mix, assembly code, static branch misprediction rates, memory access profiles for both instruction and data, etc. Although Turandot can also be used for gathering information about working set and data stream strides, we directly process acid's output for simplicity. The assembly code produced by acid is used for verification purposes.

Turandot is a PowerPC processor simulator [7]. It parses a PowerPC instruction trace and simulates the execution of the traced instructions on a user configured architecture. Since we are measuring the microarchitecture independent characteristics, we treat Turandot mainly as a trace parser, which feeds our module with each instruction's type, address, input and output. Our module then works separately to analyze the ILP, register traffic, and branch predictability for the trace. To ensure the instructions are fetched in the correct order, we turned on perfect branch prediction in Turandot and simulate the PPM branch predictor in our own module.

4.4 Statistical Analysis Methods

To compare our selected applications' characteristics in a fair and unbiased way, we need an analysis method which removes variable correlation from our data set. To this end we find it appropriate to follow in the footsteps of Eeckhout et al. [3] and use Principal Components Analysis (PCA). PCA is ideal because it allows us to eliminate any variable correlations which would otherwise skew our similarity analysis, while additionally providing a technique for reducing dataset dimensionality with controlled information loss. The second aspect is less critical to our work than it was to Eeckhout et al.'s because we are dealing with only six representative intervals rather than tens of thousands of intervals representing a complete trace of all applications, but it is still a useful property for creating a simple comparison metric.

PCA can best be summarized as a measure of variance for a dataset containing data for many variables. Each member of the the dataset is called a case, and PCA will provide a way to measure the relative similarity of the cases, based on the variances of the variables across the cases. In our analysis, each case represents one of our traced, SimPoint-selected intervals. (Note in our example we have traced only one interval for each program thread: two Doom, two Warcraft, one PBRT, and one MESA for a total of 6 intervals.) Each variable represents one of the microarchitecturally-independent characteristics detailed in section 4.1.

By creating Principal Components (PCs) from linear combinations of the original variables, we remove the effect of correlations between the variables. This makes it possible to analyze the data without fear that it has been skewed by any inter-variable relationships. Initially, we form as many PCs as there are original variables. By calculating the variance in the dataset accounted for by each PC, it is then possible to eliminate from further consideration those PCs which do contribute significantly to the total system variance. This is how we can reduce dataset dimensionality while quantitatively controlling information loss. By examining the way in which each variable contributes to the retained PCs, in terms of the coefficient associated with a variable in a given PC's formative linear combination, it is possible to give a meaningful interpretation to the PCs in terms of the original microarchitectural characteristics. Further details of the PCA process are described below.

As in Eeckhout et al. [3], we put all the data into a matrix, with each row representing a traced interval, and each column representing a variable (i.e. a characteristic). Each variable column is normalized to a mean of 0 and a variance of 1. The purpose of this is to avoid variables with large variance due to their innate scale having undue influence on our results. The resulting matrix is then operated on by the PCA. The output of PCA is another matrix, with each row representing a case and each column representing a Principal Component. We will refer to the PC in the first column as PC1, and the PC in the second column as PC2, etc. The PCs are ordered according to their variance in descending order. Consequently, we know PC1 provides the most information about the variance within our data set, followed by PC2, and so on. By setting a desired retained variance lower-bound threshold we can eliminate the majority of the PCs. For example, from the PCA of our six traces (PCAnalysis I), we select the first 2 PCs because they contribute to at least 70%of the variance of all the variables.

We normalize the selected PCs to a have variance of 1 to complete the removal of any variable correlation effects. We can then represent each case (or traced interval) as a vector, with each retained PC as a component of the vector. Euclidean distance can then be used to compare their similarity. Unlike Eeckhout et al. [3], we do not further the process with clustering techniques on our PCA data; instead we use the selected Principal Components to compare the representative intervals directly.

To provide further insight into the domain space and analysis technique we perform a second PCA analysis (PCAnalysis II). This time we incorporate not only our six new traces, but also data from fifteen of the cases studied in Eeckhout et al. [3]. These additional cases were gathered from the SPECCPU2000 benchmark suite. We choose to include all the floating point benchmarks so as to provide a wide spread of program behavior domain space against which we can compare our own traces. To produce valid comparisons between the two data sets, we choose a subset of 41 of the original characteristics which we believe should not affected by the differences in our methodology as compared to Eeckhout et al. [3]. Specifically, we were not able to directly compare working set and some instruction mix statistics. In this case, our PCA indicates that we should retain the first 4 PCs, which account for 70% of the variance across all the variables.

It is important to realize that the similarity metric provided by PCA is not absolute, but instead a relative measure. That is, the distance between the same data point pair in two different PCAs will be different if the other cases included in the PCA are changed. This is because each PCA provides a metric only in terms of those cases which are included in that PCA. The power of PCA is that it allows us to determine which cases are most or least similiar, while simultaneously gaining insight into which variables are the root cause of said similarity.

5 Results

First we show selected results for each category in section 4.1 and compare those results individually. Then we perform a PCA (PCAnalysis I) to show the composite similarities across our six threads. In the final PCA (PCAnalysis II), we include results from Eeckhout et al. [3].

5.1 Instruction Mix

Figure 1 shows a breakdown of the instruction mix used by each of our six threads. By far integer operations are the most common for all threads. MESA and PBRT include the largest amount of floating point operations - much more than either of the game threads. This may be due to the use of the graphics processors by the game programs. Probably for the same reason, the games make a much greater use of vector and cache control operations as well.

5.2 Instruction-level Parallelism (ILP)

Figure 2 shows that all threads exhibit relatively large amounts of ILP. The differences seem relatively small when considering the 32B window size, but as the window size grows, the Doom threads seem to hit a limit, indicating a relatively lower level of ILP than any of the other 4 threads.

5.3 Register Traffic

Register traffic characteristics include the average number of input operands to an instruction, the average degree of use, and the register dependency distance. The average number of input operands was similar for each thread, ranging from 1.30 for Doom 3 Thread 2 to 1.57 for MESA. The average degree of use varied slightly more from 1.11 for Doom 3 Thread 1 to 1.73 for PBRT.

The register dependency distance results are shown in Figure 3. Once the distance exceeds 2, the shape of the bar graphs and their relationship is very similar, with a small switch between MESA and PBRT in the < 8 column. It should be noted that there are still a fair number of references that have a register dependency distance greater than 64 for all of the threads, especially Doom 3 Thread 1, which seems to reach a limit somewhere just above 40%.

5.4 Working Set

The working set comparison shown in Figure 4 illustrates some of the biggest differences between the applications. Doom 3 Thread 2 is clearly performing most of the memory accesses in both the instruction and data caches. Also, Doom 3 Thread 2 and MESA clearly have significantly larger working set sizes than any of the other threads.

5.5 Data Stream Strides

Figure 6 shows the relationship between global and local data strides for each thread. Doom 3 Thread 1 has a majority of its global load strides at 64 or less, while the other threads have only 60% of their global load strides at 4K or less. The majority of global store strides are 64 or less, with MESA standing out as having global store strides that keep rising. Almost 80% of local load strides

are 64 or less for all programs, but to reach the 80% mark for local store strides requires a stride of 512 or less.

5.6 Branch Predictability

All of the threads are very predictable as demonstrated in Figure 5. The GAg predictor gets the highest misprediction rates because there are more conflicts in this predictor. Doom 3 Thread 2 is least predictable of the threads by a relatively large amount, but is still very predictable with its worst misprediction rate just above 1%.

5.7 PCA I: Doom, Warcraft, PBRT, MESA

Figure 7 shows the comparison of the vectors representing 6 threads; one each for PBRT and MESA, and two each for Doom and Warcraft. Only the comparison in the PC1 and PC2 domain space is shown here because they account for at least 70% of the variability according to the PCA. The coefficients of each PC are shown in Figure 8. Because two of the components capture such a high amount of the variability, the relative "difference" between these applications is captured in the distance they are apart in the graph.

Our results indicate that Doom 3 and Warcraft III are the least similar of our traced applications. Even the way work is divided between the major threads appears to be disparate: while both Warcraft threads exhibit very similar characteristics, the two Doom threads are furthest apart when plotted in PC1/PC2 space. This dichotomy likely springs from the huge difference between the Doom 3 threads in terms of their use of vector operations as well as their widely ranging working set sizes.

MESA, PBRT, and both Warcraft threads appear to be nearly identical with regard to our first principal component, and more divergent in terms of the second. This reflects a similarity between the programs' characteristics which are assigned a large coefficient value by the PCA. Compared to Doom 3 and MESA, PBRT is most similar in its behavior to WarCraft III. However, Doom 3's behavior is not similar to either PBRT or MESA. One possible explanation is that Doom 3 succeeds in implementing almost all graphical computations on the GPU, and is making heavy use of vector operations.

PCA enables us to see relative values for several characteristics of a particular thread by inspecting the coefficient graph. As an example, consider MESA. It has a very high PC2 value, which means it has low misprediction rates, high levels of ILP at larger window sizes, large numbers of input operands and degree of use, many floating point operations but few vector or cache operations, small global store strides, large local strides, and small working set sizes other than data32B. (All considered relative to the other threads included in the analysis.) The sign of the coefficient means larger or smaller raw data values, and the size of the coefficient indicates the degree of largeness or smallness.

5.8 PCA II: Including SPEC2000, Alpha-ISA Data

Because of the relative nature of PCA, we thought it useful to attempt to compare a number of other benchmarks to those used in this study. We also wanted to see how our results compared to those in the previous literature, specifically those presented in Eeckhout et al. There were several concerns when attempting this comparison. First, the data we have from Eeckhout et al. is aggregate data over many intervals of 100M instructions, where our data is over one large 600M interval. To compensate for this we removed characteristics that were influenced by the interval size, and depended on the Simpoint methodology to have identified a representative point in our applications. Second, we are analyzing traces from a dynamic run of the applications in question, where Eeckhout et al. are simulating large portions of the programs. Third, we are using a different instruction set architecture, and compiler than Eeckhout et al. In the final analysis, we determined that these latter two issues dominate the comparison below. This determination is largely because of the great disparity between our MESA and Eeckhout's MESA in the PCA. We did compute the distance in the four PC dimensions between our game threads and all of the other data points. It is true that the open source thread closest to three of the game threads is PBRT. The exception is Doom 3 thread 1 which is relatively far away from all other threads, but closest to one of Eeckhout's data points for a SPEC benchmark. This leads us to believe that, if the differences that do exist are taken into consideration, PBRT performance has the most potential of these programs for shedding light on micro-architecture enhancements for these games (particularly Warcraft). The lack of compatibility between the MESA results highlights the benefit of continued research for charcteristics and methodologies that can be used across ISAs, compilers, and ideally, tracing and simulation methodologies.

6 Summary and Conclusions

This paper identifies salient characteristics of two games, Doom 3 and Warcraft II, and compares them to two open-source programs, PBRT and MESA of the SPEC2000

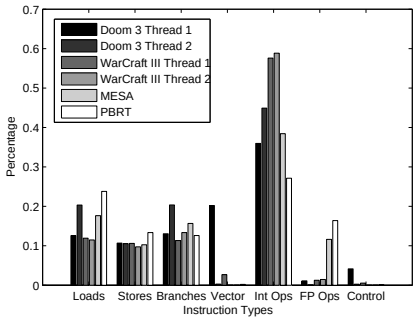

Figure 1: Instruction Mix

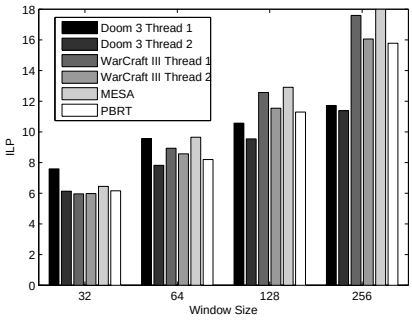

Figure 2: Instruction Level Parallelism

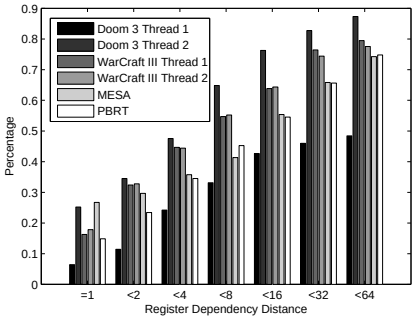

Figure 3: Register Dependencies

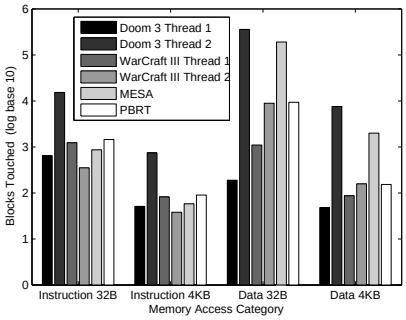

Figure 4: Working Set Comparison

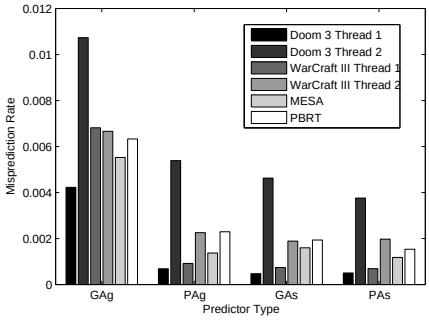

Figure 5: Prediction Comparison

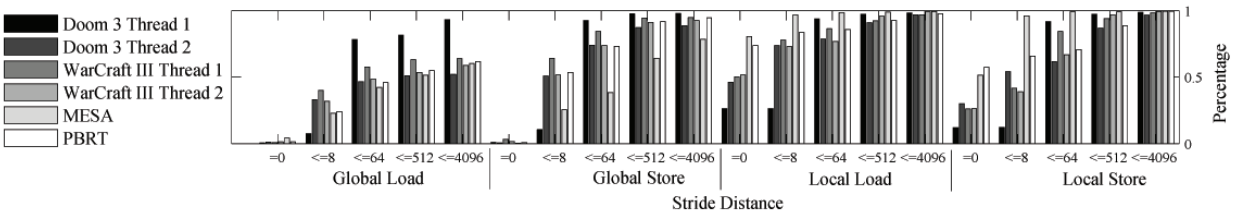

Figure 6: Data Stream Stride Comparison

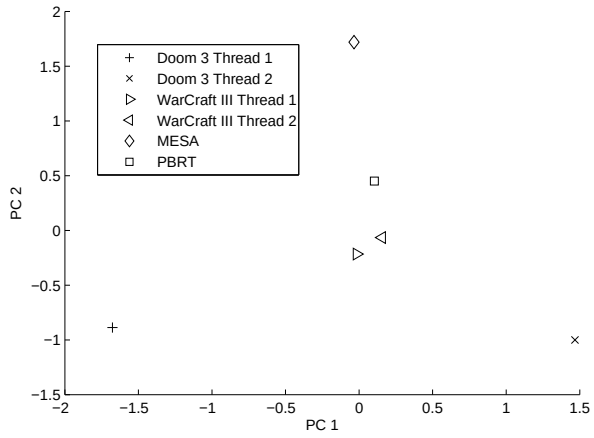

Figure 7: PCAnalysis I: Doom, Warcraft, PBRT, and MESA

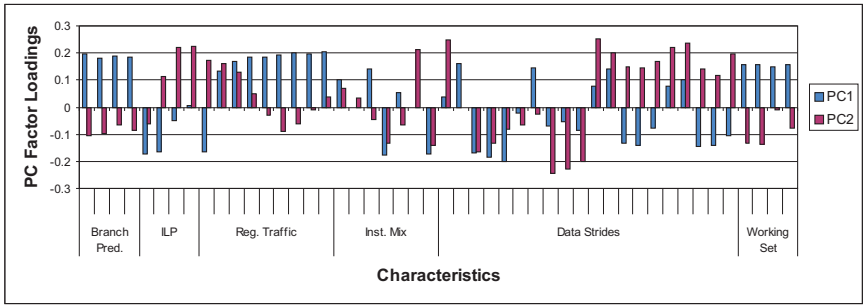

Figure 8: PCAnalyis I: Coefficients for PC1 and PC2

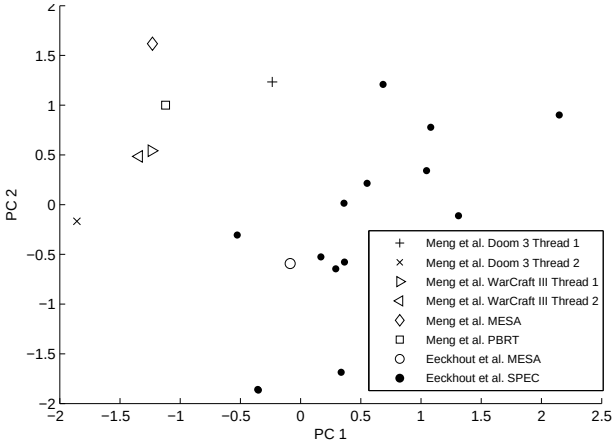

Figure 9: PCAnalysis II: Combined Meng, Eeckhout data PC1 and PC2 only

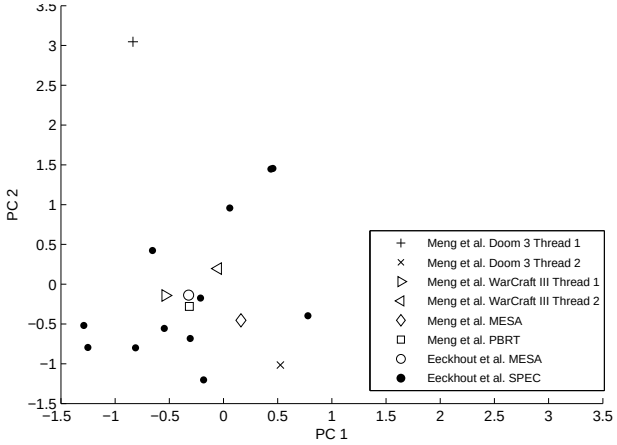

Figure 10: PCAnalysis II: Combined Meng, Eeckhout data PC3 and PC4 only

benchmark suite. Since the game programs are multi-threaded, we analyze two threads from each game and one thread from the others. Detailed comparisons are given on each of six main categories of micro-architectural characteristics. A few highlights are: 1) The game programs use more specialized instructions than MESA and PBRT, probably because they are better optimized for the architecture. 2) All threads show a fair amount of ILP, but as window size grows the Doom threads seem to hit a limit around 10. 3) All of the threads have register dependencies that exceed 64 instructions. 4) The Doom threads are highly dissimilar, while the Warcraft threads are very similar. For example, nearly all the memory blocks touched by Doom are touched by thread 2, in both the instruction and data accesses, while the memory blocks touched by the Warcraft threads are more comparable and much smaller in number. 5)Global load and local store strides are bigger, in general, for all threads than global store and local load strides. 6)All threads are very predictable in theory, with Doom 3 Thread 2 being the least predictable. Our Principal Component Analysis shows the most commonality between PBRT and the Warcraft III threads, and the largest disparity between MESA and the Doom 3 threads.

In the final analysis, we determine that the games are different enough from our two initial candidates to warrant analyzing more games and more open-source programs to track down similarities. There is almost an infinite variety within the game genre itself, including first-person shooters vs. real-time strategy vs. sports simulations, different types of game logic, different amounts of physical simulation, different amounts of GPU offload, different amounts of optimization, etc.

The similarity between Mesa/PBRT and the games, compared to the diversity among the game threads, suggests just how noisy this space is. This difference seems similar to the variance among games, and PBRT, especially, seems much more similar to these games than any other SPEC benchmarks, suggesting that optimizing for Mesa/PBRT is at least going to lead architects in the right direction. It certainly illustrates the need to understand this space better and develop better benchmarks. No one benchmark or even a small set will be representative of the game space: it will require many benchmarks.

Games are fundamentally important in the marketplace and a major design driver for CPUs, not just GPUs, yet academic architects are mostly unable to study them because we lack benchmarks and hence lack any insight into games' requirements. This is a first step to addressing this shortfall and elucidating some research directions that would allow academics to play a relevant role in this expanding market.

7 Future Work

Future work in this area falls into three general categories; work on evaluating more games and more benchmarks to discover similarities, refining characteristics and methodology to allow comparisons across ISAs and compilers, and expanding these results by using full-speed dynamic runs with performance counters to collect pertinent data.

Acknowledgments

This work is supported in part by the National Science Foundation under grant nos. NSF CAREER award CCR-0133634, and CNS-0340813, and a grant from Intel MRL. Great appreciation goes to Lieven Eeckhout for the use of his SPEC2000 data and many clarifications. We would also like to thank Yingmin Li and Karthik Sankaranarayanan for assisting us with Turandot, and Greg Humphreys for his helpful input.

8 Appendix A: Raw Data of Microarchitecturally-independent Characteristics

See Figure 11.

9 Appendix B: BBV file format

A BBV file contains per interval and per basic block statistical information of each thread.

Each line represents an interval. After a T as the beginning flag, each line is divided into several word blocks in the format :B:N, each word block shows the number of instructions executed in this basic block during the current interval. B is the basic block ID and N is the number of instructions for this basic block. The basic block information is sorted in ascending order according to the block ID. A typical line in BBV with interval 100:

T:1:34 :3:12 :4:54

10 Appendix C: AmberBBV Specifications

AmberBBV entitles the user to generate a BBV file on the fly, with a specified interval length. The BBV file can then be used for phase analysis to generate simpoints. AmberBBV configures and generates BBV files per each thread and is named in the same way as the trace files. Each BBV file contains lines of intervals with basic block statistics.

AmberBBV contains 3 modules:

Interface with the amber kernel, which initialize the AmberBBV according to the user commands and also accept trace data from the amber kernel.

Trace Parser takes in the trace data from the amber kernel, analysis it based on the opcode, gather statistic information for each thread, and pass the information to BBV tracker. The statistical information and its usage are as follows: 1. Per thread number of instructions caught by amber: If the user specified the number of skipping instructions for a thread, the program should use the per thread information to determine when to start writing the trace file. It is also used for printing out the tracing status on the fly. 2. Per basic block number of instructions caught by amber. The BBV tracker is the consumer for this information.

BBV Tracker is based on BBtracker released by Calder et al. [11], but is extended for multi-threaded programs. It builds a hash table for each basic block, with the starting pc as the key value. Each time the trace parser encounters an end of a basic block, it will send the basic block level instruction count to BBV tracker, and BBV tracker will add them to the BBV statistic counter in the current interval. If a basic block turned out to be crossing two intervals, it will automatically be split in BBV tracker. So the BBV tracker is accurate. Note that in [11], the basic

block instruction count will always be added to the current interval, so it is an approximation. In most cases, this doesn't affect the phase analysis. But when a basic block has a significant size compared with the interval size, this may introduce more error to the phase analysis.

11 Appendix D: Turandot Modifications

Turandot is modified to capture microarchitecturally-independent characteristics. Four versions of Turandot is generated, each one is responsible to capture one type of information, such as working set, ILP, register traffic and branch predictability. We insert the code into the main loop of Turandot's simulator. We are using Turandot mainly as a parser of the PowerPC instruction set. Every clock cycle, the decoded instruction queue is checked for newly fetched instructions. Since the instructions are in order at this stage, we are able to feed our analysis code with the instruction in sequence. No-ops are ignored. Turandot provides information for instruction address, register input and output, memory access address, and instruction type. With these information, we are able to process our analysis.

References

[1] Kyle Bennet. The official doom 3 hardware guide. Jul. 2004.

[2] Chris Connolly. Almost cinematic graphics : nvidia geforcefx 5600. page 7, Apr. 2003.

[3] Lieven Eeckhout, John Sampson, and Brad Calder. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *IISWC05*, Oct. 2005.

[4] Apple Computers Inc. Computer hardware understanding developer tools. Jan. 2004.

[5] Greg Kasavin. Doom 3 for pc review. Aug. 2004.

[6] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS04)*, Mar.

[7] Jaime H. Moreno and Mayan Moudgill. Turandot users's guide. In *IBM Research Report RC 21968*, February 2001.

[8] Erez Perelman, Greg Hamerly, and Brad Calder. Picking statistically valid and early simulation points. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2003.

[9] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. Measuring program similarity: Experiments with spec cpu benchmark suites. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS05)*, pages 10–20, Mar. 2005.

[10] Matt Pharr and Greg Humphreys. *Physically-Based Rendering: From Theory to Implementation*. Elsevier Science and Technology Books, 2004.

[11] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2001.

	BPRED (misprediction rate)				ILP (avg. number of independent instructions per window)			
	Gag	Pag	Gas	Pas	32	64	128	256
doom 3 thread 1	0.004224	0.000686	0.000474	0.000503	7.58533	9.55899	10.5689	11.7205
doom 3 thread 2	0.010733	0.005386	0.004624	0.003758	6.128785	7.82047	9.540499	11.38442
warcraft 3 thread 1	0.006814	0.000918	0.000737	0.000692	5.95724	8.93112	12.565	17.5899
warcraft 3 thread 2	0.006659	0.002254	0.001886	0.001971	5.97989	8.56697	11.5422	16.0542
mesa	0.005524	0.001368	0.001598	0.001177	6.44567	9.6521	12.9059	17.9952
pbrt(bunny)	0.006327	0.002293	0.001932	0.001534	6.15474	8.19613	11.2902	15.7759

	REGISTER TRAFFIC		(percentage of different dependency distances)						
	avg. input ops	avg. degree of use	depend = 1	depend < 2	depend < 4	depend < 8	depend < 16	depend < 32	depend < 64
doom 3 thread 1	1.54373	1.11064	0.0642429	0.114196	0.242145	0.331299	0.426299	0.459657	0.484081
doom 3 thread 2	1.30091	1.60463	0.252041	0.34507	0.475498	0.648178	0.762522	0.827465	0.873198
warcraft 3 thread 1	1.50526	1.35857	0.16254	0.323945	0.446511	0.546999	0.638445	0.764147	0.794743
warcraft 3 thread 2	1.46951	1.25123	0.178508	0.327606	0.444181	0.552168	0.643468	0.743939	0.775531
mesa	1.57126	1.68694	0.267281	0.296671	0.357519	0.413172	0.553824	0.65825	0.742441
pbrt(bunny)	1.52947	1.73086	0.148496	0.23419	0.345068	0.452246	0.545253	0.656173	0.747932

	ITYPES (%)						
	loads	stores	branch	altivec	interger ops	fp operations	cache control
doom 3 thread 1	12.57	10.68	13.05	20.21	35.95	1.1	4.15
doom 3 thread 2	20.32	10.52	20.35	0.28	44.9	0.06	0.27
warcraft 3 thread 1	11.93	10.61	11.32	2.68	57.61	1.25	0.53
warcraft 3 thread 2	11.49	9.74	13.34	0	58.85	1.46	0
mesa	17.64	10.27	15.68	0	38.44	11.62	0
pbrt(bunny)	23.8	13.36	12.59	0.22	27.14	16.38	0.1

	STRIDES	Gload (% of specified Memory Access Distances)				Gstore (% of specified Memory Access Distances)				
	0	<=8	<=64	<=512	<=4096	<=0	<=8	<=64	<=512	<=4096
doom 3 thread 1	0.5102	7.4096	78.3212	81.5715	93.2643	0.9219	10.5586	92.4983	97.6337	97.8472
doom 3 thread 2	1.2206	33.1945	46.627	51.0464	52.1425	0.4665	50.8981	74.0818	87.3327	88.7922
warcraft 3 thread 1	0.8474	39.9531	57.7622	63.1764	64.0141	3.2708	64.1187	84.5048	94.5	94.7796
warcraft 3 thread 2	1.3316	31.8895	48.4893	53.5669	58.8218	1.4902	51.7388	73.8627	91.2545	92.549
mesa	4.2063	22.9594	42.2306	51.5677	60.3572	0.2616	25.5789	38.4108	64.2329	78.5461
pbrt(bunny)	1.2749	24.145	45.9682	54.9862	61.6682	0.8427	53.657	73.1678	91.8525	94.6598

	STRIDES	Lload (% of specified Memory Access Distances)				Lstore (% of specified Memory Access Distances)				
	0	<=8	<=64	<=512	<=4096	<=0	<=8	<=64	<=512	<=4096
doom 3 thread 1	26.32	26.3842	93.873	97.4651	98.2998	12.1243	12.1998	91.7457	97.4464	98.774
doom 3 thread 2	46.0928	73.7885	78.6785	90.9348	96.8622	30.0853	54.1574	61.7604	86.9003	96.7671
warcraft 3 thread 1	50.3038	78.0084	86.4338	92.4641	96.8718	26.1156	41.7132	84.3531	94.1118	98.303
warcraft 3 thread 2	51.7037	73.0486	76.8157	95.8123	992,773	26.3609	38.9443	66.8618	96.8182	99.5029
mesa	80.6163	96.8509	98.5389	99,958	99.3652	51.6004	96.176	99.2597	99.268	99.617
pbrt(bunny)	73.7397	83.6428	85.9552	92.615	97.7828	57.3421	65.9362	70.7674	88.7316	99.6446

	WORKING SET (number of working sets with specified sizes)			
	Istream 32B	Istream 4KB	D stream 32B	D stream 4KB
doom 3 thread 1	648	51	189	48
doom 3 thread 2	15341	743	358936	7555
warcraft 3 thread 1	1235	82	1099	87
warcraft 3 thread 2	352	38	8890	158
mesa	864	58	191359	1999
pbrt(bunny)	1448	90	9362	153

Figure 11: Raw Data