Insight into Application Performance Using Application-Dependent Characteristics

Waleed Alkohlani[1(✉)], Jeanine Cook[2], and Nafiul Siddique[1]

[1] Klipsch School of Electrical and Computer Engineering, New Mexico State University, Las Cruces, USA
{wkohlani,nafiul}@nmsu.edu
[2] Sandia National Laboratories, Albuquerque, USA
jeacook@sandia.gov

Abstract. Carefully crafted performance characterization can provide significant insight into application performance and can be beneficial to computer designers, compiler and application developers, and end users. To achieve all the benefits of performance characterization, the characterization must incorporate a comprehensive set of characteristics that affect performance and can be measured with minimal perturbation from the underlying micro-architecture. To this end, we advocate the use of *application-dependent* characteristics that allow general conclusions to be drawn about the application itself rather than its observed performance on a specific architecture. In our prior work [7], we introduced a set of application-dependent characteristics and showed that they are consistent across architectures. In this work, we present an efficient characterization methodology that incorporates a more comprehensive set of application-dependent characteristics. We also explain in detail how these characteristics can be used to reason about and gain insight into application performance. Finally, we report characterization results on SPEC MPI2007 and Mantevo benchmarks. To our knowledge, this is the first work to present application-dependent characterization results for SPEC MPI2007 and some of the new Mantevo benchmarks.

1 Introduction

If carefully crafted, application performance characterization can provide valuable insight into performance and significant benefits to a wide range of users from hardware designers to application developers and end users. Architecture designers can use application performance characterization to quickly define an optimal initial baseline architecture for a given application or workload. Performance characterization also helps reveal code optimization opportunities for application developers and aids end-users in selecting the platform(s) that result in optimal performance. Furthermore, application benchmark developers use characterization to choose benchmarks that are representative of a particular domain and/or to compare benchmarks and determine their (dis)similarity. Finally, performance characterization can be used to provide insight into why an application performs the way it does on a particular architecture.

S.A. Jarvis et al. (Eds.): PMBS 2014, LNCS 8966, pp. 107–128, 2015.
DOI: 10.1007/978-3-319-17248-4_6

To achieve these and other benefits of performance characterization, the characterization must incorporate a comprehensive set of characteristics that affect performance and the measurements must be done in a micro-architecture-independent fashion. By using a comprehensive set of important performance characteristics, a more complete picture of application performance can be drawn. Therefore, in this work, we present and advocate the use of application-dependent (i.e., micro-architecture-independent) characteristics that allow general conclusions to be drawn about the application itself rather than its observed performance on a specific architecture. In other words, because they are the characteristics of the application that realize the observed performance, application-dependent characteristics help us understand the *fundamental cause* of the observed performance on a specific architecture.

In our prior work [7], we introduced a set of application-dependent characteristics and showed that they are consistent across architectures. In this work, we present an efficient characterization methodology that incorporates a more comprehensive set of application-dependent characteristics including spatial and temporal locality, memory usage and memory footprint, branch predictability, instruction mix, as well as characteristics related to ILP (instruction-level parallelism). To allow these characteristics to be measured quickly and in a micro-architecture independent manner, we define all characteristics such that they are easily obtainable using dynamic binary instrumentation (DBI). By using *only* DBI, our methodology does not depend on slow (possibly inaccurate) simulators and is, therefore, faster.

Although the idea of micro-architecture-independent characteristics has been explored in prior studies, the methodology and metrics presented in this paper are defined and used differently as illustrated below and in Sect. 5. Further, the set of measured characteristics (metrics) defined is more comprehensive than prior studies [12–14,20] and includes new metrics.

Workload characterization has been primarily used to understand the behavior of applications on specific platforms and to understand the similarity of benchmarks within or across benchmark suites. In this work, we define a characterization method that can be applied in a wider context. In particular, we show how to use the results of application-dependent characterization to

- reason about and gain insight into application performance
- intuitively understand how performance characteristics map to machine characteristics
- aid in benchmark comparison and/or selection.

Additional contributions of this work include (1) a comprehensive set of application-dependent metrics that includes new performance metrics, and (2) detailed performance characterization data for benchmarks that have not been characterized before as well as others that have only been lightly studied.

2 Methodology and Characteristics

In this section, we present our application-dependent performance characteristics and metrics and show how they can be used to gain insight into application performance. Our aim is to define a minimum number of characteristics that maximally

capture an application's unique and diverse behavior. We also briefly describe how to use characterization results to compare applications or to select benchmarks for a particular study. The application-dependent characteristics are classified into *general* and *memory* characteristics as described below.

2.1 General Characteristics

Dynamic Instruction Mix

The dynamic instruction mix provides information about the types and ratios of instructions executed by an application and can be used to gain a high-level understanding of what the application needs in terms of the type of execution units. To support CISC (e.g., x86) instructions that perform multiple operations, we decompose each instruction into its single operations (ops) such as *add*, *load*, or *store* ops. All the operations performed by a program are then grouped into the following five categories: (1) Loads, (2) Stores, (3) FP Ops, (4) Int Ops, and (5) Branches. These categories are chosen to correspond to the different execution units that may be implemented in a micro-architecture. Additionally, for each category, we capture a frequency distribution of the distance separating two same-type ops measured in number of instructions. Such a distribution helps us understand how particular execution units are stressed. For example, having multiple FP execution units can improve performance if FP ops occur in bursts (i.e., one after another). The distance distributions contain 513 distances or bins that start from zero to a maximum distance of 511, with the last entry representing distances larger than or equal to 512. Figure 1a shows an example distribution of the distances between load ops for the *104.milc* benchmark. The figure shows that load op pairs that follow each other (i.e., distance of 1) represent approximately 18 % of the total loads in the benchmark.

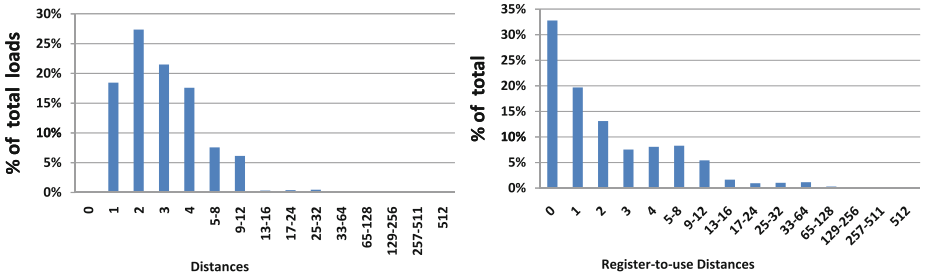
Instruction Dependence

We characterize the dependence between instructions using the register dependence distance, which is the distance measured in number of dynamic instructions between the instruction writing or producing a specific register and the instruction reading or consuming it. For each application, we capture a frequency distribution of register dependence distances. This characteristic is indicative of the amount of ILP (Instruction-Level Parallelism) inherently present in the application and indicates whether the application can utilize increased processor issue width, more in-flight instructions (i.e., larger window), or more execution units. For example, if an application exhibits tight register dependence distances, the opportunities to execute multiple instructions in parallel become limited, which in turn leads to decreased performance. In contrast, an application with long dependence distances will perform better on wide-issue processors. Figure 1b shows the register dependence distance distribution for the *CloverLeaf* benchmark. The figure shows that *CloverLeaf* has tight dependence distances; a register is written and then read by the same instruction (i.e., distance of 0) 32 % of the time while a register is written and read by the next instruction (i.e., distance of 1) about 20 % of the time.

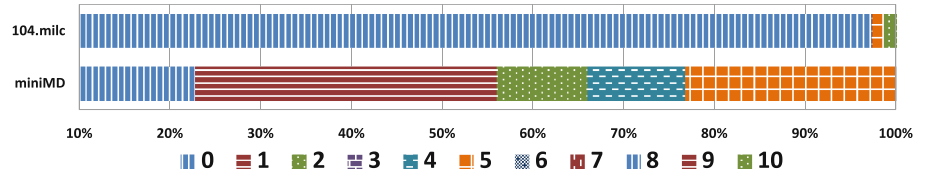
Conditional Branch Predictability

Conditional branch predictability is measured for a given application using a metric called *branch transition rate* [11]. Branch transition rate measures how often a branch switches direction between taken and not taken during execution. Branches are easily predictable if they do not change direction often or if they switch direction most of the time. Branches that have a transition rate of around 50 % are the most difficult to predict. We classify branches into 11 groups (0–10) based on their transition rates: 0–5 %, 5–10 %, 10–15 %, 15–20 %, 20–30 %, 30–70 %, 70–80 %, 80–85 %, 85–90 %, 90–95 %, and 95–100 %. Class 0 corresponds to the percentage of branches that transition 0–5 % of the time; class 1 corresponds to the percentage of branches that transition 5–10 % of the time and so on.

An application that has mostly class 0 or class 10 branches requires only a simple branch predictor and will likely experience a low misprediction rate. In contrast, an application characterized by primarily class 5 branches requires a more sophisticated predictor and will more likely have a higher misprediction rate. Figure 1c shows the percentage of branches in each branch transition rate class for the *miniMD* and *104.milc* benchmarks. *MiniMD* has a high percentage of hard-to-predict branches (Classes 4 and 5) while *104.milc* has mostly easy-to-predict branches (Class 0). Therefore, *miniMD* is likely to have a higher branch misprediction rate than *104.milc* (see Sect. 4).



(a) *104.milc* Load-Load Distance Histogram (b) *CloverLeaf* Reg. Dependence Histogram



(c) % of Branches in Each Branch Transition Class for *miniMD* and *104.milc*

Fig. 1. Example general characteristics

Computational Intensity

Computational intensity is the ratio of floating-point operations to memory accesses and is a commonly used characteristic for floating-point scientific applications. Computational intensity is an indirect measure of *data movement*.

Because moving a piece of data is typically much slower than doing an operation on it, application and algorithm developers strive to achieve higher computational intensities. Reducing data movement also reduces energy.

Average Instruction Size
The average size (in bytes) of instructions executed by an application can aid in understanding how an application utilizes a given fetch width and whether a wider fetch width is needed. This is particularly useful for CISC (e.g., x86) instructions that vary in size, affecting both the fetch and decode stages of a processor pipeline. To achieve optimal performance, the block of bytes (code) fetched on every cycle must at a minimum contain a number of instructions equal to the processor width (i.e., dispatch and commit width). We measure a distribution of instruction sizes from which we calculate the average size.

Average Basic Block Size
A basic block is a single-entry, single-exit sequence of code. Measured in number of instructions, basic block sizes are indicative of the amount of ILP available to exploit which, in turn, informs fetch width and is correlated to branch frequency. Since taken branches typically cause what is called a *fetch bubble* in a processor pipeline, an application with small basic blocks (i.e., high rate of branches) may experience frequent fetch bubbles and thus experience a decreased fetch rate. We measure a frequency distribution of the dynamic basic block sizes and calculate the average.

2.2 Memory Characteristics

Due to the dominance of the memory system in affecting performance, understanding the inherent memory characteristics of an application is key to understanding its performance. To this end, we define a comprehensive set of memory characteristics and metrics as described below.

Data Working Set Size
The working set size determines the memory size required for an application and it is defined as the total number of unique memory bytes touched by the application during its execution. The working set size (or *data intensiveness*) helps us understand the memory demands of an application and has been found to be the biggest differentiator between real applications and benchmarks [18].

Timeline of Memory Usage
This performance metric captures the size of new memory used by an application as its execution progresses in time. Starting from the beginning of execution and for every interval of one billion instructions, we track and record the total number of new and unique memory bytes touched by the application. Besides knowing the periods of execution at which the application accesses new memory, the memory usage timeline may be used to identify phases of execution. It has been shown in [15] that the working set captured for execution intervals can be an effective phase detection method.

Figure 2 shows an example timeline for the *HPCCG* benchmark. The y-axis shows the size of new memory used as a percentage of the benchmark's total working set size, and the x-axis represents execution progress. As illustrated in the figure, the entire working set size of the *HPCCG* benchmark is accessed within the first 4 % of execution; 58 %, 36 %, and 6 % of the working set size is accessed in the first, second, and forth percent of execution, respectively. This also suggests that after 4 % of execution elapses, *HPCCG* goes into a *single* execution phase for the remainder of execution. Note that it may well be that an application initializes all of its data structures (i.e. accesses all its memory) at the beginning of execution. In such a case, the memory usage timeline can not provide useful information about execution phases (see Sect. 6).

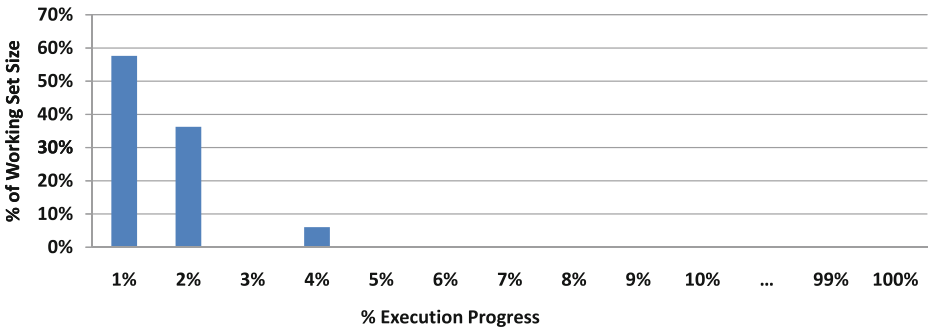


Fig. 2. Memory usage timeline for *HPCCG*

Average Requested Memory Size
This metric measures the average number of bytes read/written per memory operation, indicating the average data size used by the application. This can be useful when used with computational intensity to determine, on average, the amount of data being moved per floating-point operation. Note that depending on their types, memory instructions can read/write a widely varying number of memory bytes. Therefore, knowing the number of memory operations must be complemented by knowing the number of bytes those operations read or write.

Temporal and Spatial Locality
To mitigate the high latency of accessing memory, modern micro-architectures feature small and fast cache memories that hold frequently-accessed data closer to the processor. All caches work by exploiting the *locality of reference* exhibited (to varying extents) by all applications. There are two types of locality: *temporal locality* which is the reuse over time of a data item from memory, and *spatial locality* which is the use of data items in memory near other recently used items. By carefully analyzing an application's temporal and spatial locality, not only can we understand how effectively the application utilizes a given cache organization, but we can also reason about the optimal cache configuration for the

application. Our approach to achieving this goal starts by capturing a frequency distribution of the application's memory-reuse distances.

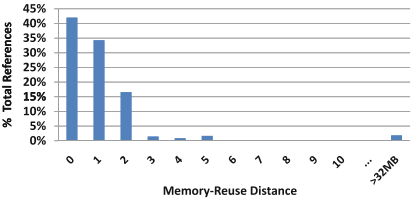
A memory-reuse distance (MRD) is defined as the distance measured in number of *unique* memory blocks accessed between two accesses to the same block. In all of our experiments, the maximum tracked MRD is 32 MB, which corresponds to a cache size of 32 MB. Using 16-byte, 32-byte, 64-byte, and 128-byte memory block sizes, we capture one MRD distribution for each block size. Note that these block sizes correspond to four potential cache line sizes. Since higher levels of cache typically store either data or instructions while lower levels of cache store both, we capture separate MRD distributions for data references, instruction references, and unified (both data and instruction) references.

We now illustrate how MRD distributions are used to characterize an application's spatial and temporal locality. Note that the conclusions drawn from the examples below are only a small sample of the conclusions that can be drawn from the data. Figure 3a shows a portion of the unified MRD distribution for the *HPCCG* benchmark. The x-axis represents the distance in number of unique 64-byte block accesses between two accesses to the same 64-byte block, and the y-axis represents the percentage of the total memory references.

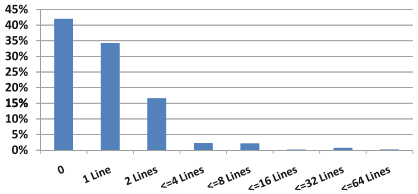
The goal of characterizing an application's spatial locality is to help us understand how effectively and quickly the application consumes the data available to it in a cache block. To achieve this and at the same time visualize spatial locality, we plot the points from the MRD distribution that correspond to short memory-reuse distances; zero through 64 (Fig. 3b). In other words, we determine the percentage of memory references that reuse data from the same block (line) after n accesses to other blocks, where $n = \{0, 1, 2, 4, 8, 16, 32, 64\}$. Other studies [12–14,20] capture spatial locality only for a distance of zero by considering only successive references. We believe that using a window of n references intuitively provides more accurate spatial locality information but is computationally more complex.

As shown in Fig. 3b, about 42 % of the references in *HPCCG* immediately reuse the same line (i.e., distance of 0), and around 34 % of references reuse the same line after one access to a different line (i.e., distance of 1). Figure 3d illustrates how *HPCCG*'s spatial locality changes over different block sizes. Within the maximum distance of 64 line accesses, 91 %, 96 %, 98 %, and 99 % of references are spatially local using 16-, 32-, 64-, and 128-byte blocks, respectively. Note that in an n-way set-associative cache, there is a possibility that the intermediate block accesses are to the same set (see discussion below), which may cause a block to be evicted by the time it is referenced again. Thus, it may be more accurate to look at spatial locality for short distances (e.g., 2, 4, and 8) that correspond to the cache associativity of interest. For example, Fig. 3d shows that the percentage of references spatially local within a distance of 2 is 70 %, 89 %, and 93 % for block sizes of 16, 32, and 64 bytes, respectively. As also seen in Fig. 3d, *HPCCG*'s spatial locality improves only slightly by increasing the block size from 64 to 128 bytes. From this, we can conclude that the optimal cache line size for exploiting *HPCCG*'s spatial locality is 64 bytes.

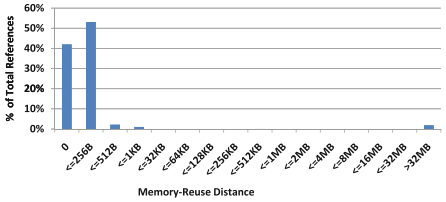
To visualize temporal locality, the distances on the x-axis of the MRD distribution are grouped into bins that correspond to potential cache sizes. The first



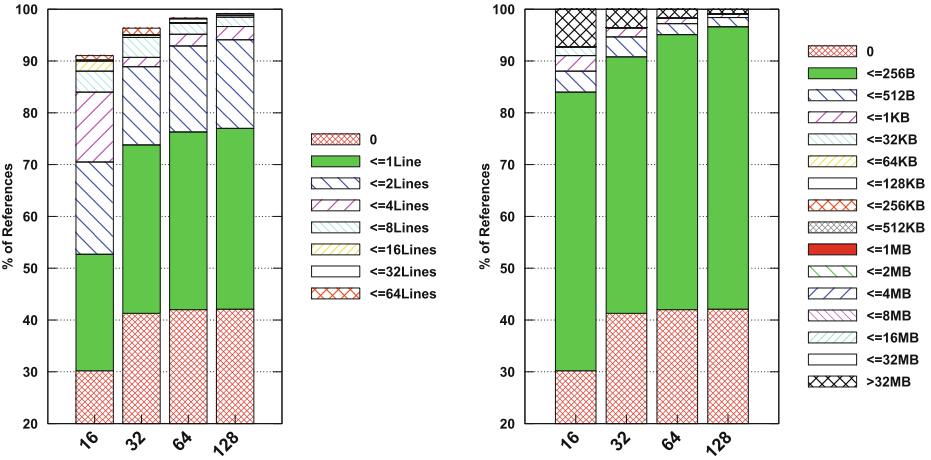
(a) Portion of *HPCCG*'s MRD Histogram



(b) Spatial Locality for *HPCCG*



(c) Temporal Locality for *HPCCG*



(d) *HPCCG*'s Spatial Locality (left) & Temporal Locality (right) for 4 Block Sizes

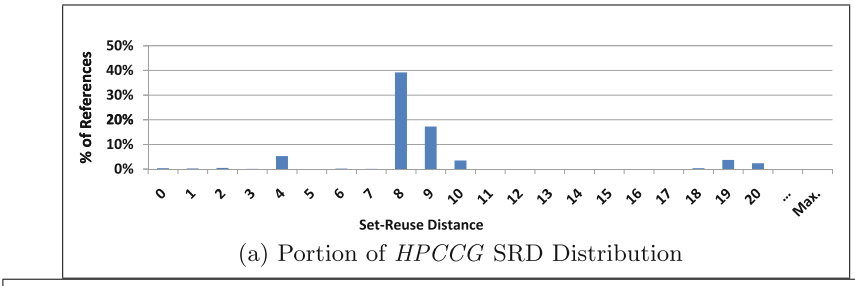
Fig. 3. Temporal and spatial locality examples

four distance bins are set to 0, 4, 8, and 64 *times* the line size. The rest of the bins go from 32 KB up to 32 MB, doubling each time. Figure 3c shows the temporal locality plot for *HPCCG* based on 64-byte blocks and unified references.

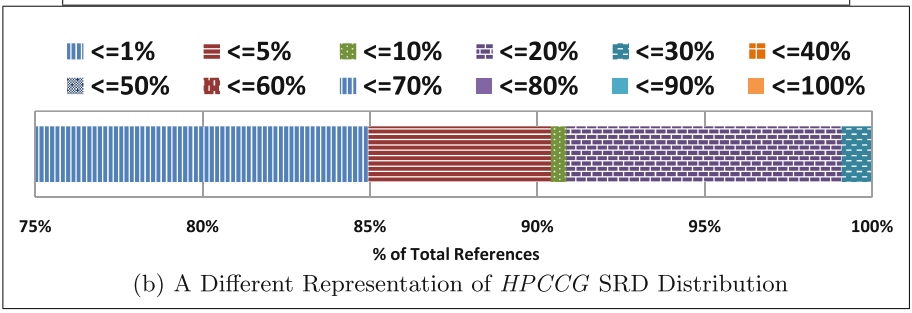
The figure shows that 95 %, 97 %, and 98 % of references are temporal within the distances of 256 B, 512 B, and 1 KB, respectively. This implies that a 1 KB cache is large enough to keep 98 % of references temporally local within the cache. Figure 3d shows how *HPCCG*'s temporal locality changes over different cache line sizes. For example, the percentage of references that are temporal within 1 KB is 91 %, 96 %, 98 %, and 99 % for 16-, 32-, 64-, and 128-byte blocks, respectively.

The above temporal and spatial locality analysis assumes that the target cache is fully associative. However, in an n-way set associative cache, the block accesses

that occur between two accesses to the same block can be to the same set, which may cause a block to be evicted by the time it is re-accessed. For caches with a high degree of associativity, which are typical of lower-level caches and closely approximate fully-associative caches, our above analysis is valid and is confirmed using actual measurements (see Sect. 4). However, for low associative caches, it is important to look at the access patterns of cache sets. To this end, we capture a frequency distribution of the set-reuse distances, where a set-reuse distance (SRD) is the number of sets accessed between two accesses to the same set. To capture the SRD distribution, assumptions must be made about the size of the cache, the size of a cache line, and the number of ways in a cache set. In all our experiments, the cache size is assumed to be 32 MB. We use four cache line sizes (16, 32, 64, and 128 bytes) and four associativities (2, 4, 8, and 16 ways). One SRD distribution is captured for every unique combination of line sizes and number of ways.



(a) Portion of *HPCCG* SRD Distribution



(b) A Different Representation of *HPCCG* SRD Distribution

Fig. 4. Set-reuse distances for *HPCCG*

Figure 4a shows a portion of the SRD distribution for *HPCCG*. In capturing this distribution, the 32 MB cache is assumed to be 8-way set associative with 64-byte lines. As shown in Fig. 4a, about 40 % of references re-access the same set after accessing eight other sets (i.e., distance of 8). It can also be seen that around 67 % of references have a set-reuse distance of less than or equal to 10. This suggests that a set is frequently re-accessed within a short period of time. This may lead to more conflict misses provided that a low-associative cache is used and that the MRD distribution shows a high ratio of references with long MRDs compared to SRDs. That is, *different* blocks within a set are frequently accessed within a short period of time, which increases the likelihood of conflict

misses. In Fig. 4b, we group the set-reuse distances into bins that represent distances as a percentage of the total sets in the 32 MB cache. As illustrated in the figure, around 85 %, 90 %, and 99 % of total references re-access the same set after 1 %, 5 %, and 20 %, respectively, of sets are accessed.

2.3 Selection and Comparison of Benchmarks

In addition to gaining insight into performance and reasoning about hardware resources optimal for performance, the application-dependent characteristics described above can also be used to select an appropriate set of benchmarks for a particular study or to determine the (dis)similarity among benchmarks. For example, if one is interested in studying branch behavior or evaluating branch predictors, they need to choose benchmarks with diverse branch predictability characteristics. On the other hand, if evaluating memory system configurations or studying memory behavior is of interest, the benchmarks with the most diverse memory characteristics should be considered.

To compare benchmarks, the metrics used to measure the application-dependent characteristics for each benchmark can be grouped into a vector that can be called *the performance vector*. For example, the percentage of each of the five categories in the instruction mix and the percentage of references in each bin of the memory-reuse distance distribution can be included in the performance vector. The performance vectors of different benchmarks can then be normalized and compared using a simple distance measure.

3 Experimental Setup

In this section, we briefly describe the platforms and tools used to capture the application-dependent characteristics described in Sect. 2 as well as the benchmarks used in this study.

Platforms
All of our experiments are conducted on a Dell cluster that includes eight nodes, each of which runs the Scientific Linux (version 6.3) operating system [4] and has 48 GB of available RAM. Each node contains two six-core Intel Xeon X5670 processors that are clocked at 2.93 GHz. While all the cores share a 12 MB 16-way L3 cache, each core has a 32 KB 4-way L1 instruction cache, a 32 KB 8-way L1 data cache, and 256 KB 8-way L2 unified cache. A cache line is 64 bytes in all the levels of cache. Each of the Intel Xeon X5670 processor cores implements the Westmere-EP micro-architecture which features: (1) a 4-way superscalar out-of-order execution pipeline, (2) a 128-entry re-order buffer, and (3) three integer, three floating-point, and four address generation units.

Tools
We capture the application-dependent characteristics described in Sect. 2 using dynamic binary instrumentation (DBI) tools that we developed in-house using Pin [17]. The slowdown caused by DBI depends on the type of analysis performed

and the number of dynamic instructions instrumented. However, DBI is still orders of magnitude faster than simulation and there exist techniques such as sampling to effectively speed up the execution of instrumented binaries.

Capturing the memory-reuse and set-reuse distance distributions (see Sect. 2) is nontrivial and can cause extreme slowdowns. To capture these reuse distances, a FIFO(First-In-First-Out) queue is typically used to hold memory references and for every new reference encountered during execution, the queue is searched for a prior occurrence of the reference to determine a reuse distance. We implement three optimization methods to speed up our DBI tool. First, we limit the size of the FIFO queue by restricting the maximum reuse distance to 32 MB which is sufficient to study the behavior of most modern caches. Second, we implement the FIFO queue using a balanced binary tree to achieve much faster search and update times. Finally, rather than instrumenting the entire benchmark binary, we use representative sampling [9,10] to select a limited number of representative samples. Then, the instrumentation is applied only to the selected samples. For each benchmark, up to ten 100-million-instruction samples are identified using the PinPoints methodology [19] which is based on the well-known SimPoint tool [24]. In [7], we show that the reuse distributions measured with and without sampling are statistically similar at 95 % confidence.

Using our optimized tools and for all the benchmarks listed in Table 1, it took approximately two weeks to capture all the characterization data on the 8-node platform described above. Finally, the PapiEx [3] tool is used to obtain counts from the on-chip hardware performance counters

Table 1. List of benchmarks used

Suite	Benchmark	Lang.	Application domain
SPEC MPI2007	104.milc	C	Quantum chromodynamics
	107.leslie3d	Fortran	Computational fluid dynamics
	113.GemsFDTD	Fortran	Computational electromagnetics
	132.zeusmp2	C/Fortran	Computational fluid dynamics
	137.lu	Fortran	Computational fluid dynamics
Mantevo MiniApps	miniFE	C++	Unstructured Implicit Finite Element
	HPCCG	C + +	Unstructured implicit finite element
	miniMD	C + +	Molecular dynamics
	miniXyce	C + +	Circuit simulation
	CloverLeaf	C/Fortran	Hydrodynamics

Benchmarks

Table 1 shows a list of all the benchmarks used in this study. Although all are parallel benchmarks, we execute them serially for the purposes of this work. All benchmarks are built using compilers from the Gnu Compiler Collection(GCC) [1] and are drawn from the following benchmark suites:

1. **SPEC MPI2007** (version 1.1) is a benchmark suite from System Performance Evaluation Corporation (SPEC) containing thirteen MPI-parallel,

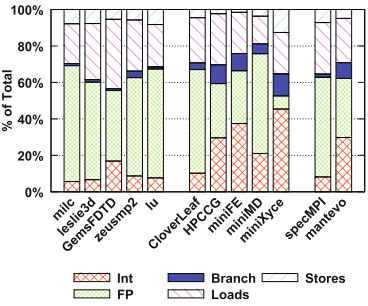
floating point, compute intensive benchmarks [5]. We select five benchmarks (Table 1) that are the only benchmarks that can be executed serially.
2. **Mantevo MiniApps**, developed at Sandia National Laboratories, are small self-contained proxies of real scientific applications used in the lab [2]. At the time of doing this study, version 1.0 of the Mantevo suite contained seven MiniApps. Of these seven MiniApps, we use five (Table 1) and exclude two (*miniGhost* and *CoMD*) that we could not successfully run with our tools. The problem sizes of the selected MiniApps are manually configured such that the number of instructions they execute is similar to that of the SPEC MPI2007 benchmarks (i.e., few trillion instructions per benchmark).

4 Results

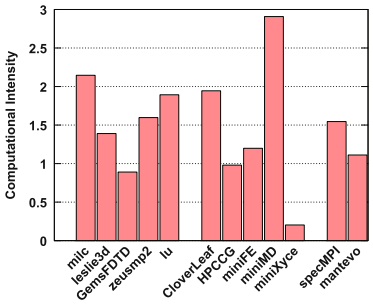
In this section, we present measured application-dependent characteristics for all the studied benchmarks. We also show performance data from on-chip counters on the platform described in Sect. 3. We select and show only the counts that help in interpreting the application-dependent characterization data.

Instruction Mix and ILP Characteristics
Figure 5a shows the instruction mix for each of the studied benchmarks. On average, SPEC benchmarks execute more floating-point instructions (55 % vs 32 %), more loads (28 % vs 24 %), and slightly more stores (7 % vs 5 %) than Mantevo benchmarks. On the other hand, Mantevo benchmarks execute more integer operations (30 % vs 8 %) and more branch instructions (9 % vs 2 %). However, unlike SPEC benchmarks, Mantevo benchmarks exhibit more diversity in their instruction mixes. For example, *CloverLeaf* and *miniMD* have high ratios of floating-point operations, *miniXyce* has the highest ratio of integer operations, and both *HPCCG* and *miniFE* have a more even distribution of integer and floating-point operations. In general, all the SPEC benchmarks will likely benefit from more floating-point execution resources while the Mantevo benchmarks will benefit from a mix of more floating-point, integer, and branch execution resources. Although both benchmark suites have relatively similar ratios of memory instructions, understanding their optimal memory resources requires the understanding of their memory access patterns and other memory characteristics that are presented later in this section.

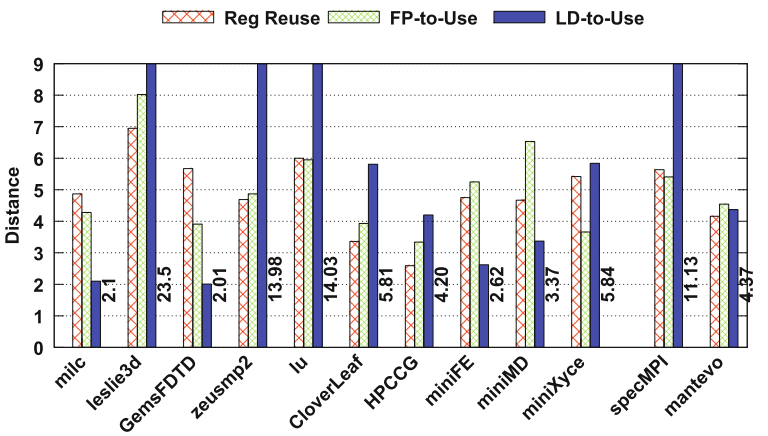
Figure 5c shows the average register dependence distance as well as the average distance between a load or a floating-point instruction to their consumer instruction. On average, Mantevo benchmarks have shorter register dependence distances (4.1 vs 5.6), shorter FP-to-use distances (4.5 vs 5.4), and much shorter load-to-use distances (4.4 vs 11.1). Also, on average, Mantevo benchmarks have smaller basic blocks than SPEC benchmarks (11 vs 33 instructions). From the above, we can conclude that SPEC benchmarks, on average, exhibit more inherent ILP (instruction-level parallelism) than Mantevo benchmarks. The long distances between load operations and their consumers in SPEC benchmarks suggests that small memory latencies may be effectively hidden through out-of-order execution.



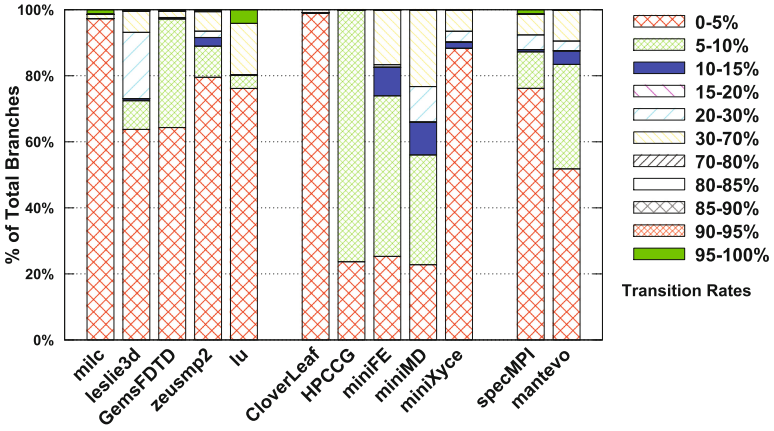
(a) Instruction Mix



(b) Computational Intensity



(c) Register Dependence, FP-to-Use, and Load-to-Use Distances (in # of Instrs)



(d) % of Branches in Each Branch Transition Class

Fig. 5. Select general characteristics of SPEC MPI2007 & Mantevo Mini Apps

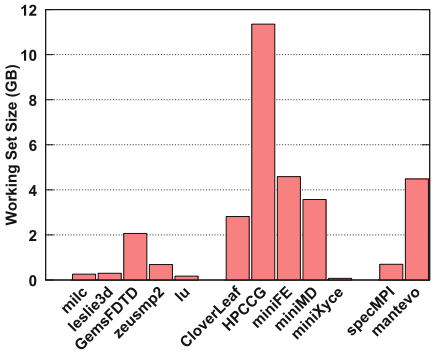
On the other hand, depending on their memory access patterns and the likelihood of experiencing cache misses, benchmarks with short load-to-use distances may require code optimization to hide memory access latencies.

Branch Predictability
Figure 5d shows the benchmarks' branch predictability using their branch transition rates (Sect. 2). For each benchmark, the figure shows the percentage of branches in each transition rate class. As described in Sect. 2, branches with high or low transition rates are more easily predictable than branches with around 50 % transition rates. Almost all benchmarks have predominantly easy-to-predict branches and thus their measured branch misprediction rates are less than 1 % as seen in Fig. 9b. However, *miniMD* has the most diverse branch predictability and the highest ratio of hard-to-predict branches. Therefore, it experiences the highest branch misprediction rate (11.4 %), which can be a serious performance bottleneck given that 5.5 % of all instructions in *miniMD* are branches.

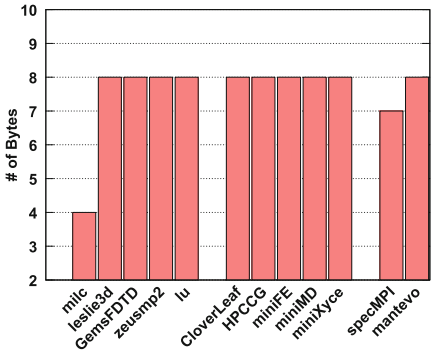
Computational Intensity
Figure 5b shows the computational intensity of all the studied benchmarks. As can be concluded from their instruction mix, SPEC benchmarks, on average, have higher computational intensities than Mantevo benchmarks. However, the Mantevo benchmarks show more diversity in computational intensity with *miniMD* being the most computationally intensive of all benchmarks and *miniXyce* the least. Note that with the exception of *milc*, memory instructions in all benchmarks read or write 8 bytes of data on average (Fig. 6b).

Data Working Set Size and Usage
Figure 6a shows the data working set size (data intensiveness) for all studied benchmarks. On average, Mantevo benchmarks have much larger working set sizes than SPEC benchmarks (4.5 GB vs 0.7 GB). As noted in Sect. 3, all benchmarks are configured such that they execute a similar number of instructions. With a working set size of 2 GB, *GemsFDTD* is the only SPEC benchmark that accesses more than 1 GB of data. The benchmark *HPCCG* has the largest working set size (11 GB) and *miniXyce* has the smallest (73 MB). Since cache performance is largely dependent on temporal and spatial locality characteristics as well as the cache configuration, having larger working sets does not necessarily lead to worse cache performance. This is supported by the actual cache miss measurements shown in Fig. 9c. These measurements show that some benchmarks (e.g., *miniXyce*) with small working sets experience more cache misses than benchmarks with much larger working sets.

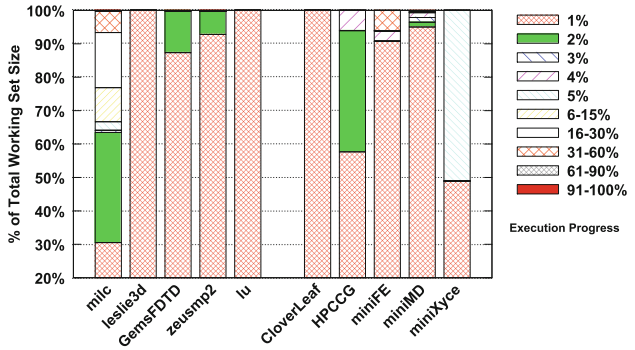
Figure 6c shows the amount of new memory accessed with respect to execution progress. With the exception of *milc* and *miniFE*, all benchmarks access their entire working sets within the first 1 to 5 percent of execution. *miniFE* accesses around 93 % of its working set within the first 3 % of execution while *milc*'s memory usage is more distributed between 1 % and 60 % of execution. This suggests that *milc* has more diverse execution phases than the other benchmarks. However, as discussed in Sect. 2, our memory usage timeline may not accurately



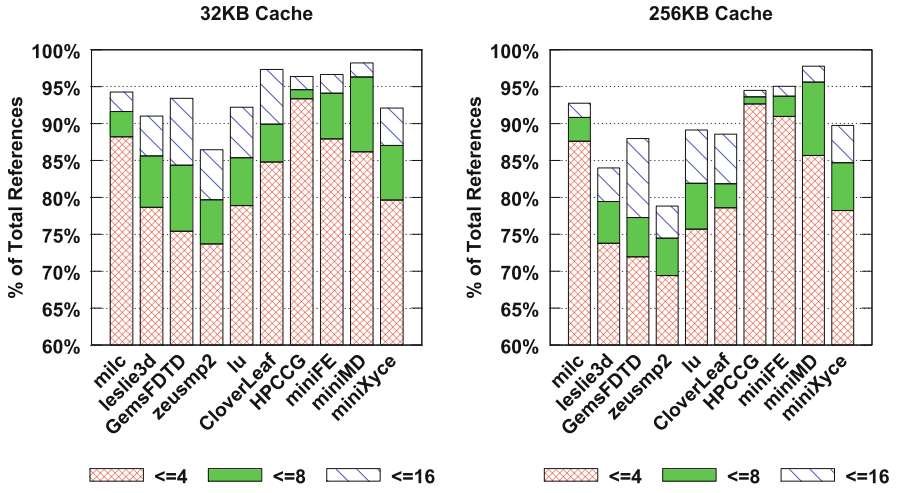
(a) Working Set Size



(b) Bytes Read/Written Per Mem Instr.



(c) Timeline of New Memory Usage



(d) Set-Reuse Distances Based on 32KB (left) and 256KB (right) Caches (64-byte lines, 8-way associativity)

Fig. 6. Select memory characteristics of SPEC MPI2007 & Mantevo mini apps

reflect execution phase behavior since benchmarks may start their execution by initializing their entire used memory. This issue will be addressed in future work.

Spatial and Temporal Locality

Spatial and temporal locality plots are presented in Figs. 7 and 8, respectively. For four different block (cache line) sizes, these plots show only the locality of data references. We describe below the locality characteristics of individual benchmarks and relate our conclusions to the actual cache miss measurements shown in Fig. 9c; cache details are in Sect. 3. To help relate conclusions to actual measurements, we capture and show in Fig. 6d the percentage of 64-byte references that have short set-reuse distances on two cache configurations that correspond to the actual 8-way 32 KB L1 and the 8-way 256 KB L2 caches implemented in the Westmere architecture (see Sect. 3 and Sect. 2).

Shown in Fig. 7, the spatial locality (i.e., the percentage of accesses reusing the same block within a small number of other accesses) of *milc* as well as of most of the other benchmarks, increases with increasing block sizes. For 16-, 32-, 64-, and 128-byte blocks, the percentage of references reusing the same block within a distance of 8 is 73 %, 83 %, 90 %, and 91 %, respectively. However, *milc*'s spatial locality increases only slightly by going from 64-byte to 128-byte blocks. As illustrated in Fig. 8, *milc* also exhibits a high degree of temporal locality with over 95 % of its 64-byte memory accesses being temporal within 4KB (64×64). Similar to its spatial locality, *milc*'s temporal locality does not significantly improve by using blocks larger than 64 bytes.

Figure 9c shows that *milc* experiences fewer L1 and L2 misses compared to the other SPEC benchmarks. This is due to its better temporal locality and better spatial locality within a distance of 8 (i.e., the L1/L2 cache associativity). However, *milc* encounters L1 and L2 misses despite its excellent temporal locality. This can be attributed to conflict misses caused by a high percentage of references re-accessing the same cache set within short distances (Fig. 6d).

Compared to *milc*, *leslie3d*, *GemsFDTD*, and *zeusmp2* exhibit less spatial locality with only 60 %, 50 %, and 50 %, respectively, of their 64-byte references being spatially local within a distance of 8 (Fig. 7). Also, of *GemsFDTD*'s, *zeusmp2*'s, and *leslie3d*'s 64-byte accesses, only 60 %, 50 %, and 50 %, respectively, are temporal within 4 KB (Fig. 8). This explains why these benchmarks experience more L1 and more L2 (except *zeusmp2*) cache misses than *milc* (Fig. 9c). The fewer L2 cache misses of *zeusmp2* can be attributed to fewer conflict misses since a lower ratio of its memory accesses reuse a recently-accessed set (Fig. 6d).

With 64-byte blocks, only about 70 % of *CloverLeaf*'s references exhibit spatial locality within a distance of 8 (Fig. 7). On the other hand, around 80 % of the memory references in *HPCCG*, *miniFE*, *miniMD*, and *miniXyce* are spatially local within a distance of 8. With the exception of *miniXyce*'s temporal locality, the spatial and temporal locality of all Mantevo benchmarks improves only slightly by increasing the block size from 64 to 128 bytes. Figure 8 shows Mantevo benchmarks have varying temporal locality for long MRDs but also high ratios of references that are temporal within short MRDs(i.e., small caches).

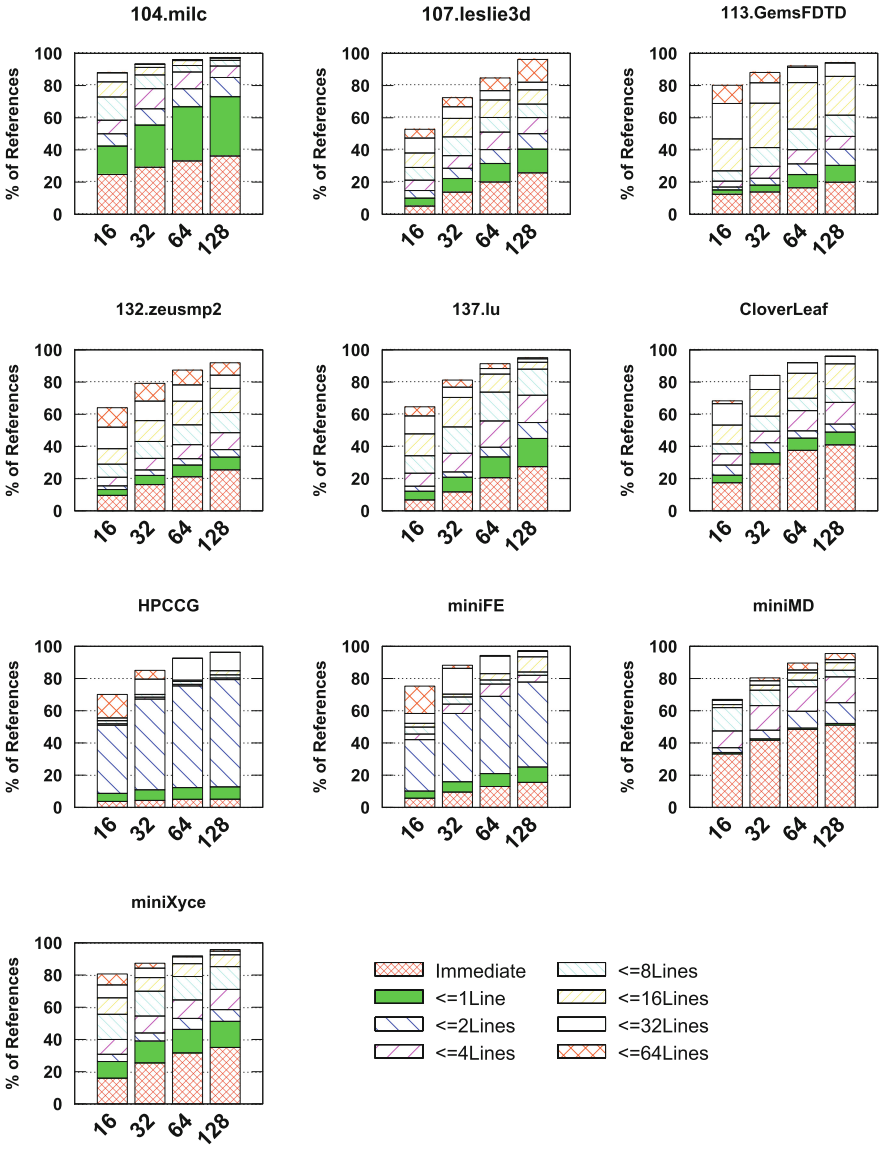


Fig. 7. Spatial locality

Because it exhibits less spatial and temporal locality compared to the other Mantevo benchmarks, *CloverLeaf* experiences more L1 cache misses (Fig. 9c). It also encounters more L2 cache misses than the other Mantevo benchmarks except *miniXyce*. As seen in Fig. 6d, both *CloverLeaf* and *miniXyce* have lower ratios of references with short set-reuse distances, which further indicates lower locality. *MiniXyce* has more L2 and L3 cache misses than all the other Mantevo

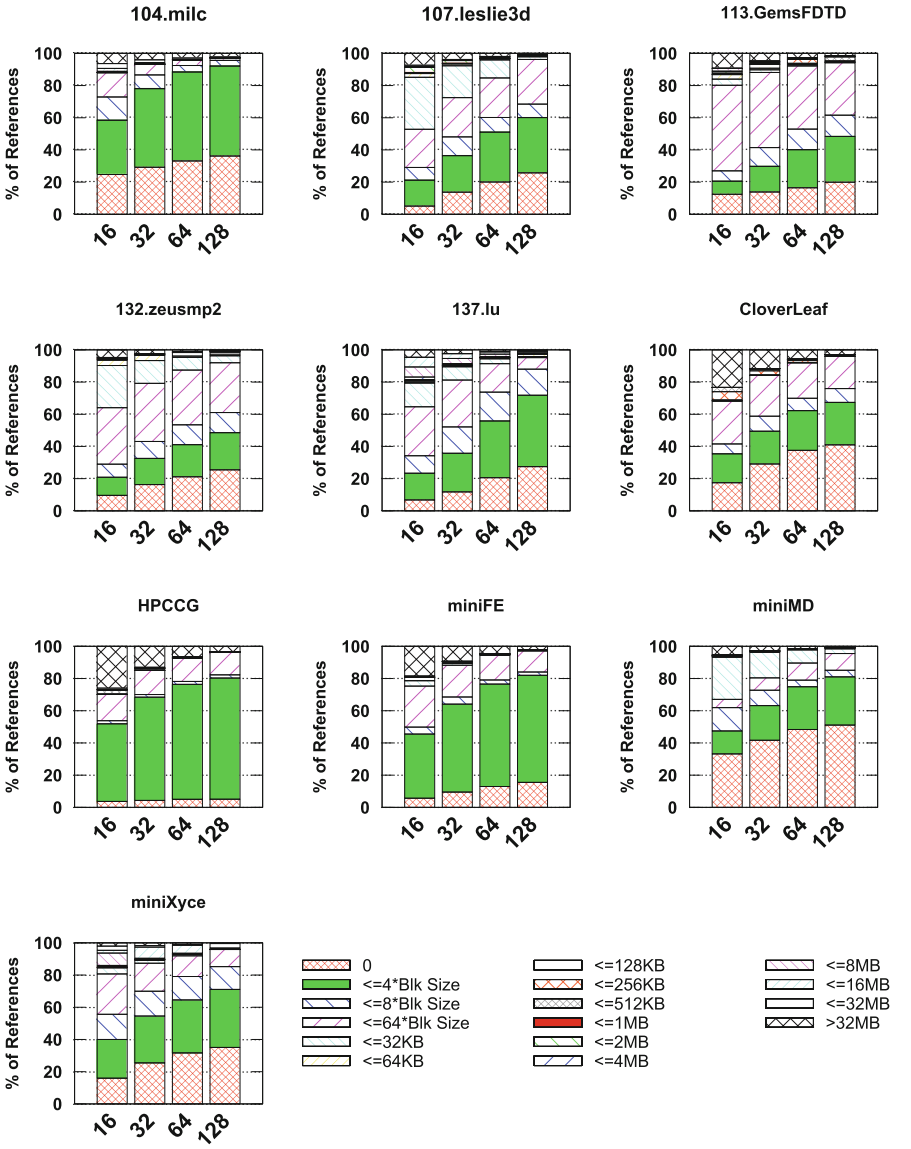
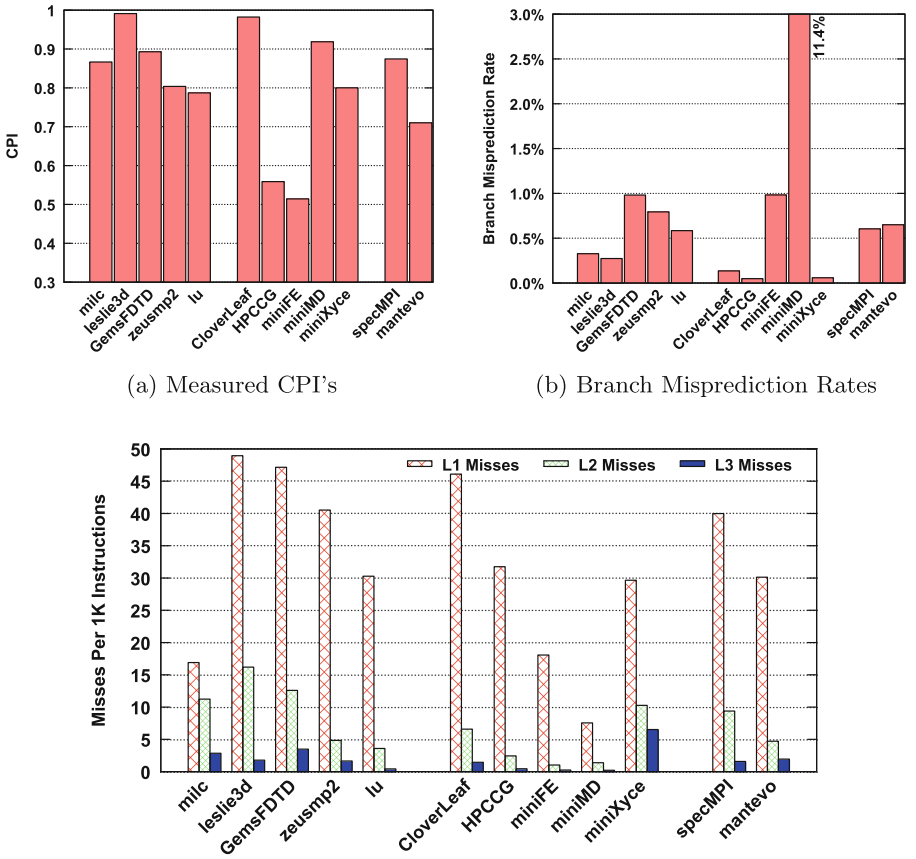


Fig. 8. Temporal locality

benchmarks because around 10 % of its references are not temporal within 256 KB or 12MB (i.e., L2 and L3 cache sizes). This also explains why most of its L2 cache misses are not satisfied in the L3 cache. On the other hand, *miniMD* has the lowest number of L1 and L2 cache misses because it exhibits the best temporal locality within 32 KB (99 % of references) and the highest ratio of references immediately reusing the same cache line.



(a) Measured CPI's

(b) Branch Misprediction Rates

Fig. 9. Performance measurements from the intel platform (Sect. 3)

Discussion of Performance Measurements

Figure 9 presents the CPI, branch misprediction rates, and cache misses measured on the Intel platform described in Sect. 3. On average, Mantevo benchmarks perform better than SPEC benchmarks (0.71 vs 0.87 CPI). The relatively low performance of SPEC benchmarks could largely be attributed to their higher cache miss rates and higher ratio of floating-point operations. However, there is substantially more variance in the CPI's of the Mantevo benchmarks. This is consistent with the fact that Mantevo benchmarks exhibit more varying application-dependent characteristics as shown earlier in this section.

Of the SPEC benchmarks, *leslie3d* shows the worst performance (highest CPI). We believe this is due to its relatively larger number of L1 and L2 cache misses. On the other hand, although *milc* has a low L1 cache miss rate, it exhibits a relatively high CPI. This may be attributed to its short load-to-use distances (Fig. 5c) with which cache access latencies can not be effectively hidden.

Of the Mantevo benchmarks, *CloverLeaf* has the highest CPI which can be attributed to its relatively high L1, L2, and L3 cache miss rates. Also, with

CloverLeaf's tight register dependence distances, cache miss penalties can not be hidden. In contrast, *miniMD* shows a relatively high CPI although it has the lowest number of cache misses. This may be largely attributed to its high branch misprediction rate. *MiniXyce's* CPI is also high and can be attributed to its relatively high L1, L2, and L3 miss rates. It also has the smallest basic blocks (5 instructions on average) and large instruction sizes (4 bytes on average) which can limit the number of instructions fetched by the processor per cycle (see Sect. 2). Finally, both *HPCCG* and *miniFE* encounter the lowest number of L2 and L3 cache misses. Because these two benchmarks exhibit relatively good ILP (long register dependence and load-to-use distances), their L1 cache miss penalties can be effectively hidden, which may explain their low measured CPI.

5 Related Work

Most prior characterization approaches use **hardware-dependent** performance metrics such as CPI or cache miss rates obtained from hardware performance counters or simulation [6,8,16,23]. The goal is to measure and understand application performance on a specific platform. Other approaches use similar hardware-dependent metrics to study benchmark similarities to find representative subsets of benchmark suites [21,22]. Besides the pitfalls of hardware-dependent characterization mentioned in [12,13], conclusions drawn from these studies only apply to the specific micro-architecture used. However, using **micro-architecture-independent** metrics as presented in this work, allows us to reason about application behavior on different machines, even those that do not exist yet.

Other studies use microarchitecture-independent characteristics such as instruction mix and memory footprint to study program similarities [12–14,20]. The primary objective of these studies is to reduce the number of benchmarks used in design space exploration and to discover programs with similar or unique program behavior within a benchmark suite. Besides being applied in a wider context, our methodology includes a more diverse set of characteristics and metrics. Also, the metrics we use are either new or *different* in that similar metrics are *defined* differently and, therefore, capture different behavior. For example, we capture branch predictability and consider larger windows for capturing ILP characteristics. Furthermore, we track a much larger number of memory references and provide a more precise definition of temporal and spatial locality to help in correlating with actual cache measurements.

6 Conclusions and Future Work

This work presents an architecture-independent methodology for characterizing application performance that is based on binary instrumentation and incorporates a diverse set of application-dependent characteristics. We report results on SPEC MPI2007 and Mantevo benchmarks. We show that SPEC benchmarks are more computationally intensive while Mantevo benchmarks have much larger memory demands. Also, Mantevo benchmarks exhibit more diverse behavior in

all dimensions than SPEC benchmarks. To our knowledge, this work is the first to present architecture-independent characterization results for SPEC MPI2007 and some Mantevo benchmarks.

In future work, we plan to enhance our approach to capture the working set size such that it can accurately be used for detecting execution phases. We also plan to extend the methodology to characterize more aspects of performance that are important in multi-threaded and parallel applications such as synchronization and data movement.

References

1. The gnu compiler collection. http://gcc.gnu.org
2. The mantevo project. http://mantevo.org/
3. PapiEx. http://icl.cs.utk.edu/mucci/papiex/
4. Scientific Linux. http://www.scientificlinux.org/
5. SPEC MPI2007 benchmark suite. http://www.spec.org/mpi2007/
6. Bird, S., Phansalkar, A., John, L.K., Mercas, A., Idukuru, R.: Performance characterization of SPEC CPU benchmarks on Intel's core microarchitecture based processor. In: SPEC Benchmark Workshop (2007)
7. Alkohlani, W., Cook, J.: Towards performance predictive application-dependent workload characterization. In: Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC 2012, pp. 426–436. IEEE Computer Society, Washington, DC (2012). http://dx.doi.org/10.1109/SC.Companion.2012.62
8. Conte, T.M., Wen-mei, W., Hwu, W.: Benchmark characterization. Computer **24**, 48–56 (1991)
9. Hamerly, G., Perelman, E., Calder, B.: How to use simpoint to pick simulation points. SIGMETRICS Perform. Eval. Rev. **31**(4), 25–30 (2004)
10. Hamerly, G., Perelman, E., Lau, J., Calder, B.: Simpoint 3.0: faster and more flexible program analysis. J. Instr. Level Parallelism **7**, 1–28 (2005)
11. Haungs, M., Sallee, P., Farrens, M.: Branch transition rate: a new metric for improved branch classification analysis. In: Proceedings of the HPCA, pp. 241–250 (2000)
12. Hoste, K., Eeckhout, L.: Comparing benchmarks using key microarchitecture-independent characteristics. In: 2006 IEEE International Symposium on Workload Characterization, pp. 83–92, October 2006
13. Hoste, K., Eeckhout, L.: Microarchitecture-independent workload characterization. IEEE Micro **27**(3), 63–72 (2007)
14. Joshi, A., Phansalkar, A., Eeckhout, L., John, L.K.: Measuring benchmark similarity using inherent program characteristics. IEEE Trans. Comput. **55**(6), 769–782 (2006). http://dx.doi.org/10.1109/TC.2006.85
15. Lau, J., Schoemackers, S., Calder, B.: Structures for phase classification. In: Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2004, pp. 57–67. IEEE Computer Society, Washington, DC (2004). http://dl.acm.org/citation.cfm?id=1153925.1154588
16. Li, S., Qiao, L., Tang, Z., Cheng, B., Gao, X.: Performance characterization of SPEC CPU2006 benchmarks on intel and amd platform. In: First International Workshop on Education Technology and Computer Science, ETCS 2009, vol. 2, pp. 116–121 (2009)

17. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, PLDI 2005, pp. 190–200. ACM, New York (2005). http://doi.acm.org/10.1145/1065010.1065034
18. Murphy, R.C., Kogge, P.M.: On the memory access patterns of supercomputer applications: Benchmark selection and its implications. IEEE Trans. Comput. **56**(7), 937–945 (2007). http://dx.doi.org/10.1109/TC.2007.1039
19. Patil, H., Cohn, R., Charney, M., Kapoor, R., Sun, A., Karunanidhi, A.: Pinpointing representative portions of large intelitaniumprograms with dynamic instrumentation. In: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 37, pp. 81–92. IEEE Computer Society, Washington, DC (2004). http://dx.doi.org/10.1109/MICRO.2004.28
20. Phansalkar, A., Joshi, A., Eeckhout, L., John, L.: Measuring program similarity: Experiments with spec cpu benchmark suites. In: IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2005, pp. 10–20, March 2005
21. Phansalkar, A., Joshi, A., John, L.K.: Analysis of redundancy and application balance in the. SPEC CPU2006 benchmark suite. In: Proceedings of the 34th annual international symposium on Computer architecture, ISCA 2007, pp. 412–423. ACM, New York (2007)
22. Phansalkar, A., Joshi, A., John, L.K.: Subsetting the SPEC CPU2006 benchmark suite. SIGARCH Comput. Archit. News **35**(1), 69–76 (2007). http://doi.acm.org/10.1145/1241601.1241616
23. Poovey, J.A., Levy, M., Gal-On, S., Conte, T.: A benchmark characterization of the EEMBC benchmark suite. IEEE Micro. **PP**(99), 1–1 (2009). doi:10.1109/MM.2009.50
24. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, pp. 45–57. ACM, New York, NY (2002)