Using Cycle Stacks to Understand Scaling Bottlenecks in Multi-Threaded Workloads

W. Heirman
T.E. Carlson
S. Che
K. Skadron
L. Eeckhout

Report 09.2011.3, September 2011

This work is funded by Intel and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT).

Using Cycle Stacks to Understand Scaling Bottlenecks in Multi-Threaded Workloads

Wim Heirman[1,3] Trevor E. Carlson[1,3] Shuai Che[2] Kevin Skadron[2] Lieven Eeckhout[1]

[1]Department of Electronics and Information Systems, Ghent University, Belgium
[2]Department of Computer Science, University of Virginia
[3]Intel Exascience Lab, Belgium

Abstract

This paper proposes a methodology for analyzing parallel performance by building cycle stacks. A cycle stack quantifies where the cycles have gone, and provides hints towards optimization opportunities. We make the case that this is particularly interesting for analyzing parallel performance: understanding how cycle components scale with increasing core counts and/or input data set sizes leads to insight with respect to scaling bottlenecks due to synchronization, load imbalance, poor memory performance, etc.

We present several case studies illustrating the use of cycle stacks. As a subsequent step, we further extend the methodology to analyze sets of parallel workloads using statistical data analysis, and perform a workload characterization to understand behavioral differences across benchmark suites. We analyze the SPLASH-2, PARSEC and Rodinia benchmark suites and conclude that the three benchmark suites cover similar areas in the workload space. However, scaling behavior of these benchmarks towards larger input sets and/or higher core counts is highly dependent on the benchmark, the way in which the inputs have been scaled, and on the machine configuration.

1 Introduction

Power efficiency has driven the computer industry towards multicore processors. Current general-purpose processors employ a limited number of cores in the typical range of 4 to 8 cores; see for example Intel Westmere-EX or Sandy Bridge, IBM POWER7, AMD Bulldozer, etc. It is to be expected that the number of cores will increase in the coming years, given the continuous transistor density improvements predicted by Moore's law, as exemplified by Intel's Many Integrated Core architecture with more than 50 cores on a chip [12].

A major challenge with increasing core counts is the ability to analyze and optimize performance for multicore systems. Computer architects need performance analysis tools and workload characterization methodologies to understand the behavior of existing and future workloads in order to design and optimize future hardware.

This paper makes the case for building *cycle stacks* to understand and analyze performance of multi-threaded workloads. A cycle stack quantifies where the cycles have gone, and provides more information than raw event rates, such as miss rates. A cycle stack is typically represented as a stacked bar with the different components showing the relative contribution of each component to overall performance. The key benefit of a cycle stack is that it provides quick insight into the major performance bottlenecks, which hints towards optimization opportunities. This is particularly interesting for analyzing parallel performance: by analyzing how the cycle stacks change with increasing core counts, one can understand whether scaling bottlenecks come from synchronization overhead, poor performance in the memory hierarchy, load imbalance, etc.

This paper presents a methodology for analyzing the performance of multi-threaded programs using cycle stacks. The methodology is simulation-based for a number of reasons. First, cycle stacks cannot be readily measured on real hardware. Existing performance counters enable measuring a large number of events, however, so far it is challenging to build accurate cycle stacks from them [8]. Second, we want both software developers and computer architects to use the methodology to study performance scalability of parallel workloads on multicore hardware. We employ the recently proposed Sniper simulation infrastructure [3] which can simulate multi-threaded workloads running on shared-memory machines at a speed in the MIPS range; validation has shown the simulator to achieve good accuracy compared to real hardware. As part of our methodology, we extend the Sniper simulator with a novel critical path cycle accounting mechanism to compute detailed cycle stacks. We present

This is the authors' version of the work. The authoritative version will appear in Proceedings of IISWC, Nov. 2011.

several case studies illustrating the value of cycle stacks for analyzing parallel performance scalability issues.

As a second part of this paper, we employ cycle stacks to analyze scaling behavior of three multi-threaded benchmark suites, namely SPLASH-2 [18], PARSEC [1] and Rodinia [4]. To deal with the large volume of data produced by this analysis, we apply Principal Component Analysis (PCA) to derive some general scaling trends. We study scaling behavior with increasing core counts and increasing data set sizes, and analyze differences in scaling behavior across benchmark suites. From this analysis we conclude that, although SPLASH-2, PARSEC and Rodinia stress similar components of the system, their scaling behavior to larger core counts and larger input sets differs. The results also suggest directions in which each suite might fruitfully be expanded to encompass a wider range of scaling behaviors.

This paper makes the following contributions:

- We propose a methodology that uses cycle stacks to analyze parallel workload performance, and PCA to derive general performance scaling trends across workloads and systems.
- We propose a novel critical path cycle accounting mechanism for computing accurate and detailed cycle stacks through simulation, which captures the impact of ILP on performance.
- We compare three prevalent benchmark suites, SPLASH-2, PARSEC and Rodinia, and analyze differences in scaling behavior using cycle stacks. Prior work, on the other hand, used only abstract workload characteristics to compare these benchmark suites, and did not provide this level of detail.
- We use cycle stacks to analyze scaling behavior with the number of cores and input data sets. We conclude that scaling behavior is highly dependent on the benchmark, the way in which the inputs have been scaled, and on the machine configuration. In addition, we point out a potential pitfall when using small input data sets for studying large multicore processors, which advocates future work in simulation methods that scale with increasing core counts and input data sets.

2 Methodology for Analyzing Multi-Threaded Workload Performance

Figure 1 gives a high-level overview of the overall framework. The input to the framework is a benchmark or a set of benchmarks. These workloads are run on a simulator to obtain cycle stacks. The simulator employed in this work is an interval simulator [3] extended with a novel critical path cycle accounting mechanism for constructing cycle stacks. The end user can readily use these cycle stacks to analyze performance and identify performance bottlenecks. If the user is interested in studying parallel performance across a set of workloads, a challenge then is how to deal with the large volume of raw performance data: for each program there is a cycle stack for each thread, and each cycle stack consists of multiple cycle components, which makes performance analysis across a set of workloads challenging. We therefore employ Principal Component Analysis (PCA), which is a statistical data analysis technique that extracts important trends from large volumes of data. PCA plots show how workloads differ among each other along the most significant dimensions. Finally, in case the output from PCA is too large to be visualized in a low-dimensional graph — which we found to be the case in our study — clustering can be performed to obtain dendrograms that represent the (dis)similarity across workloads.

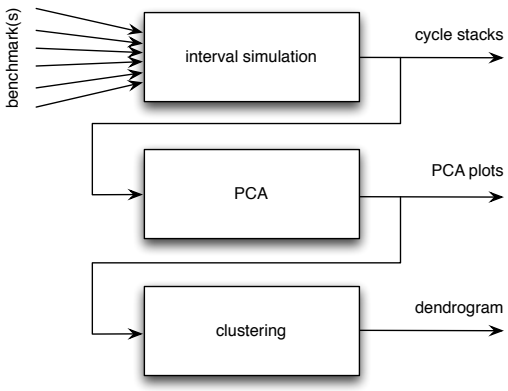

Figure 1. Overview of the methodology.

The remainder of this section presents a description of the proposed methodology. We start off with interval simulation and how we measure detailed cycle stacks. We then describe how we aggregate cycle stacks across threads in multi-threaded workloads. Subsequently, we detail PCA and clustering.

2.1 Interval simulation

Interval simulation is a recently proposed simulation paradigm for simulating multi/manycore and multiprocessor systems at a higher level of abstraction than the current practice of detailed cycle-accurate simulation [11]. Interval simulation leverages a mechanistic analytical model [9] to abstract core performance by driving the timing simulation of an individual core without the detailed tracking of individual instructions through the cores pipeline stages. The mechanistic analytical model is constructed from the underlying mechanisms of a superscalar processor core. The

foundation of the model is that miss events (branch mispredictions, cache and TLB misses) divide the smooth streaming of instructions through the pipeline into so-called *intervals*. Branch predictor, memory hierarchy, cache coherence and interconnection network simulators determine the miss events; the analytical model derives the timing for each interval. The cooperation between the mechanistic analytical model and the miss event simulators enables modeling of the tight performance entanglement between co-executing threads on multicore processors.

The multicore interval simulator models the timing for the individual cores. The simulator maintains a window of instructions for each simulated core that corresponds to the reorder buffer of an out-of-order processor, and is used to determine miss events that are overlapped by long-latency load misses. The functional simulator feeds instructions into this window at the window tail. Core-level progress (i.e., timing simulation) is derived by considering the instruction at the window head. In case of an I-cache miss, the core simulated time is increased by the miss latency. In case of a branch misprediction, the branch resolution time plus the front-end pipeline depth is added to the core simulated time, to model the penalty for executing the chain of dependent instructions leading to the mispredicted branch plus the number of cycles needed to refill the front end of the pipeline. In case of a long-latency load (i.e., a last-level cache miss or cache coherence miss), we add the miss latency to the core simulated time, and we scan the window for independent miss events (cache misses and branch mispredictions) that are overlapped by the long-latency load — second-order effects. If none of the above cases applies, we dispatch instructions at the effective dispatch rate, which takes into account inter-instruction dependencies as well as their execution latencies.

Compared to the prevalent one-IPC approach, which assumes that each core executes one instruction per cycle apart from memory accesses which are assumed to be blocking, interval simulation is slightly more complex while being substantially more accurate [3]. The key benefits over one-IPC modeling are that interval simulation models non-blocking or out-of-order execution, the impact of instruction-level parallelism (ILP), the impact of memory-level parallelism (MLP) or overlapping memory accesses, as well as second-order overlap effects (e.g., an independent branch misprediction being resolved underneath a long-latency memory access). In this work, we use the Sniper simulator [3], which implements the interval simulation paradigm in Graphite [15].

2.2 Measuring cycle stacks

We now first revisit cycle stacks before describing how we measure them through interval simulation.

2.2.1 Cycle stacks

The total cycle count for a computer program executing on a processor can be divided into a base cycle count plus a number of cycle components that reflect lost cycle opportunities due to miss events such as branch mispredictions, cache and TLB misses, synchronization overhead, etc. The breakdown of the total number of cycles into components is often referred to as a cycle 'stack' — or a Cycles-Per-Instruction (CPI) stack in case one divides cycle count by the number of executed instructions. The reason for calling it a 'stack' is because the cycle components are typically displayed as stacked histogram bars where the cycle components are placed one on top of another with the base cycle component being shown at the bottom of the histogram bar. (See Figure 3 for an example of a cycle stack.) A cycle stack reveals valuable information about application behavior on a given processor and provides more insight into an application's behavior than raw miss rates do. In fact, a cycle stack provides quick insight in the major performance bottlenecks, i.e., the largest cycle components hint towards optimization opportunities because optimizing these cycle components could lead to large performance improvements.

2.2.2 Critical path cycle accounting

As part of this work, we extended the interval model to construct more detailed cycle stacks using a novel critical path cycle accounting mechanism. Prior work in interval modeling and simulation [9, 11] computes the base cycle count, or base CPI, as the average throughput that can be achieved through a given window. This is done through Little's law: throughput is computed by dividing the number of instructions in the window by the average critical path length. The critical path length is determined by the longest chain of dependent instructions and takes into account inter-instruction dependencies as well as non-unit instruction execution latencies. As a result, the base cycle count is represented as a single cycle component in the cycle stack, lumping together different ILP effects into one component.

To gain more insight into the different mechanisms that affect the base cycle component, we introduce *critical path accounting*. This mechanism tracks which instruction types are responsible for making up the critical path through the execution of the program. This is done by breaking up the critical path into its contributors, namely the different instructions along the critical path. Then, each time the interval simulation model increments time by one cycle — which resulted in a complete cycle being attributed to the base component in the old scheme — we distribute the attribution of the current cycle over all components that make up the critical path at that time.

Figure 2 shows an example to illustrate the concept. Assume the first instruction executes at cycle 0. Instruction

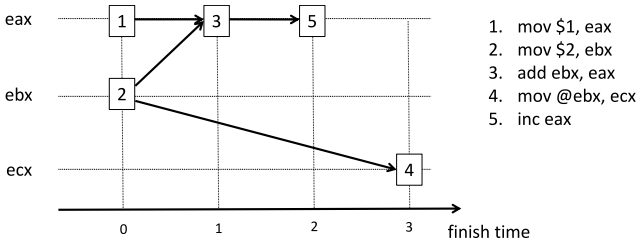

Figure 2. Critical path accounting.

two is independent so it can start its execution in the same cycle. Instruction three consumes the result of instruction one, through register `eax`. Since an integer addition has a latency of a single cycle, instruction three will finish execution in cycle 1. The critical path through all instructions thus far has a length of one cycle. Instruction four, a memory load, depends on instruction two through a dependence on register `ebx`. If this load hits in the first-level cache, it will have a latency of three cycles, and will finish execution in cycle 3. The critical path increases from one to three cycles. The critical path accounting algorithm now attributes the increase of the critical path by two cycles to the load execution unit. Finally, instruction five executes in cycle 2, and does not extend the critical path.

At this point in time, the critical path through the reorder buffer has a length of three cycles. One cycle is attributed to integer addition, while two cycles are attributed to memory load. If the interval model then decides to increment time by one cycle — assume there are no miss events happening and this cycle is accounted as a base cycle — this cycle is accounted according to the critical path, i.e., .33 cycles are attributed to the *depend-int* component while .67 cycles are attributed to the *mem-l1d* component. This proportional weighting makes that, in the absense of miss events, each cycle in the critical path will ultimately be attributed to a full cycle in the total execution.

When instructions commit and leave the reorder buffer, they are removed from the dependency graph and their contribution from the critical path is removed as well. The critical path is not recomputed when removing instructions, just as in the original interval model: we approximate the critical path length by subtracting the time stamps of instructions at both ends of the window and only perform the critical path accounting at insertion. This way, simulation speed is not compromised while preserving accuracy. The effect of this on cycle attributions is that instructions are prioritized according to program order, i.e., instructions that can execute completely in the shadow of instructions earlier in the program never have any cycles attributed to them.

2.3 Aggregating cycle stacks

Collecting cycle stacks for multi-threaded workloads yields a cycle stack for each thread. If the goal is to gain insight in general performance trends and scaling behavior across workloads and architectures, the amount of information quickly becomes overwhelming. As a first step towards analyzing the large volume of data, we first aggregate the cycle stacks to obtain a compact representation. The general intuition is that homogeneous threads, i.e., threads doing similar work, exhibit similar cycle stacks and hence they can be aggregated easily.

We start by selecting only threads/cores that do real work. Some of the benchmarks do not fully utilize all available cores; for instance, in most of the PARSEC benchmarks, the main thread only starts worker threads but does not do any computation itself. Threads that spend more than half of their time in synchronization routines such as `pthread_cond_wait` or `pthread_barrier_wait` should not affect total application performance and are therefore ignored. Next, the cycle components for all cores are added together on a per-component basis.

For heterogeneous workloads, in which different threads execute different code paths, we split up the benchmark into multiple thread groups. Each thread group corresponds to a collection of threads that do execute the same code.[1] We handle a thread group as if it were one homogeneous workload. Again, only thread groups that execute real work are selected.

2.4 Principal Component Analysis

We now have the ability to measure and aggregate cycle components for each of the workloads of interest. However, in the process of analyzing the data that we had collected for a number of benchmarks in the SPLASH-2, PARSEC and Rodinia benchmark suites, we quickly realized that the amount of data was simply too large to handle efficiently. Hence, we decided to leverage Principal Component Analysis (PCA) as a statistical data analysis technique to identify the major trends in the large volume of data. PCA [6] essentially reduces the dimensionality of a data set without losing too much information. More precisely, it transforms a number of possibly correlated variables (or dimensions) into a smaller number of uncorrelated variables, which are called the principal components. Intuitively speaking, PCA has the ability to describe a huge data set along a limited number of dimensions, and present a lower-dimensional picture that still captures the essence of the more-dimensional data set.

[1] We only consider the top-level function executed by the thread. Minor variations in control flow do not cause threads to be split into different groups.

We use PCA to analyze a data set consisting of cycle components for each of the workloads. The workloads are shown in the rows. Each row represents a benchmark with a specific input run on a specific architecture. Hence, different inputs to a benchmark show up as separate rows in the data matrix, as well as the same workload run on different architectures, e.g., with different core counts. The columns represent the various cycle components. One could view this data matrix as a p-dimensional space with each dimension being a cycle component, and each workload a point in this p-dimensional space. Because p can be quite large in practice ($p = 15$ in our study), getting insight into this high-dimensional space is non-trivial. In addition, correlation exists between the cycle components, which further complicates the ability to derive insights from the results. PCA transforms the p-dimensional space to a q-dimensional space (with $q \ll p$) in which the dimensions are uncorrelated. The transformed space provides an opportunity to understand workload (dis)similarity. Workloads that are far away from each other in the transformed space show dissimilar behavior; workloads that are close to each other show similar behavior. Having workloads in the data matrix from different benchmark suites, with different input data sets, and run on different processor architectures, enables one to understand how program behavior is affected by these factors, as we will demonstrate later in this paper.

2.5 Clustering

The end result from PCA is a data matrix with n rows (the workloads) and q columns (the principal components). As we will see in the results section, PCA enables capturing and analyzing major performance trends, however, the q-dimensional space is yet too complicated to be represented in an easy-to-understand way. In particular, analyzing and gaining insight in, say, a four-dimensional space is non-trivial. Hence, we employ cluster analysis to summarize the data even further, grouping the n workloads based on the q principal components to obtain a number of clusters, with each cluster grouping a set of workloads that exhibit similar behavior.

There exist two common clustering techniques, namely agglomerative hierarchical clustering and K-means clustering [14]. We use agglomerative clustering in this paper because it produces a dendrogram which is valuable for representing relative distances among workloads in a high-dimensional space. Agglomerative hierarchical clustering considers each workload as a cluster initially. At each iteration of the algorithm, the two clusters that are closest to each other are grouped to form a new cluster. The distance between the merged clusters is called the *linkage distance*. Nearby clusters are progressively merged until finally all benchmarks reside in a single big cluster. This clustering process can be represented in a so-called *dendrogram*, which graphically represents the linkage distance for each cluster merge. Small linkage distances imply similar behavior in the clusters, whereas large linkage distances suggest dissimilar behavior. There exist a number of methods for calculating the distance between clusters — the inter-cluster distance is needed in order to know which clusters to merge. We use average-linkage clustering in this paper, i.e., the inter-cluster distance is computed as the average distance between the cluster members.

3 Experimental Setup

3.1 Simulator configuration

As mentioned earlier, we use the Sniper parallel simulator as our simulation infrastructure. Carlson et al. [3] validated this simulator against the Intel Xeon X7460 Dunnington system and showed good absolute and relative accuracy. We configured Sniper to model a quad-socket SMP machine; see Table 1 for details. Each socket contains four cores, for a total of 16 cores in the machine. Each core is a 45 nm Penryn microarchitecture, and has private L1 instruction and data caches. Two cores share the L2 cache, hence, there are two L2 caches per chip. The L3 cache is shared among the four cores on each chip. The simulator was configured to use barrier synchronization with a quantum of 100 cycles; this is to keep the simulated threads synchronized during parallel simulation. Each thread spawned by the benchmark application is pinned to its own simulated core. Sniper is a user-space simulator so it does not model the operating system nor a scheduler, although emulation of some aspects that impact performance, such as system call overhead, have been added. Simulation speed in this configuration is around 2 MIPS, which allows us to complete the simulation of a typical benchmark used in this study in around 1 to 6 hours on a modern 8-core host machine.

3.2 Benchmarks

In this paper, we evaluate a diverse set of applications from three benchmark suites to understand and compare their performance bottlenecks. SPLASH-2 [18] is a widely used collection of multi-threaded workloads, consisting of 12 benchmarks mostly targeted towards high-performance computing. PARSEC [1], a benchmark suite jointly developed by Princeton University and Intel, is targeted towards chip multiprocessors (CMPs), includes multi-threaded workloads from emerging application domains (recognition, mining and synthesis). The Rodinia [4] benchmark suite is designed for heterogeneous computing. It includes CUDA and OpenCL implementations for the GPU platform and OpenMP implementations for multicore CPUs; we consider the OpenMP versions in this study.

Parameter	value
Sockets per system	4
Cores per socket	4
Clock frequency	2.67 GHz
Dispatch width	4 micro-operations
Reorder buffer	96 entries
Branch predictor	Pentium M [17]
Cache line size	64 B
L1-I cache size	32 KB
L1-I associativity	8 way set associative
L1-I latency	3 cycle data access, 1 cycle tag access
L1-D cache size	32 KB
L1-D associativity	8 way set associative
L1-D latency	3 cycle data access, 1 cycle tag access
L2 cache size	3 MB per 2 cores
L2 associativity	12 way set associative
L2 latency	14 cycle data access, 3 cycle tag access
L3 cache size	16 MB per 4 cores
L3 associativity	16 way set associative
L3 latency	96 cycle data access, 10 cycle tag access
Coherence protocol	MSI
Main memory	200 ns access time
Memory Bandwidth	4 GB/s

Table 1. Simulated system characteristics.

We use all the benchmarks from these three benchmark suites that ran properly on the simulation infrastructure; see Table 2 for the list of benchmarks included in this study. We failed to run some of the benchmarks because of limitations in the simulator. Table 2 also shows the inputs used for our experiments. For each benchmark, we consider two inputs — small and large — to understand how workload behavior varies with input data set size. The PARSEC benchmark suite defines a number of input set sizes, we selected `simsmall` and `simlarge`. SPLASH-2 and Rodinia do not provide multiple standard input sets, so we scaled the inputs to obtain application run times that roughly match those of the corresponding PARSEC input sizes.

The initialization part of each benchmark is ignored in our simulations. We do this by enabling the timing model only at the start of the Region of Interest (ROI), and disabling it again at the end of the ROI; the ROI is the parallel section of the workload. The PARSEC 2.1 distribution already has these ROI regions marked in the source code; we manually marked the parallel section for SPLASH-2 and Rodinia.

All benchmarks are homogeneous except for `dedup` and `ferret`. Both these applications are from the PARSEC suite and employ functional pipelining. By inspecting the source code, we know which threads execute what code and we select those thread groups that do real work for at least half the time, as described in Section 2.3. For `dedup` there is only one such thread group, executing the `compress` function. `ferret` has two thread groups, marked in the source code by `vec` and `rank`. In the graphs, we refer to these thread groups as `dedup-compress`, `ferret-vec` and `ferret-rank`, respectively.

Benchmark	'small' input size	'large' input size
SPLASH-2		
`barnes`	16384 particles	32768 particles
`cholesky`	tk25.O	tk29.O
`fmm`	16384 particles	32768 particles
`fft`	256K points	4M points
`lu.cont`	512×512 matrix	1024×1024 matrix
`lu.ncont`	512×512 matrix	1024×1024 matrix
`ocean.cont`	258×258 ocean	1026×1026 ocean
`ocean.ncont`	258×258 ocean	1026×1026 ocean
`radiosity`	–room –ae 5000.0 –en 0.050 –bf 0.10	–room
`radix`	256K integers	1M integers
`raytrace`	car –m64	car –m64 –a4
`volrend`	head-scaleddown2	head
`water.nsq`	512 molecules	2197 molecules
`water.sp`	512 molecules	2197 molecules
PARSEC 2.1	*simsmall*	*simlarge*
`blackscholes`	4096 options	65536 options
`canneal`	100k elements	400k elements
`dedup`	10 MB	184 MB
`facesim`	372,126 tetrahedra	372,126 tetrahedra
`ferret`	16 queries, 3,544 images	256 queries, 34,973 images
`freqmine`	250000 transactions	990000 transactions
`raytrace`	480×270 pixels	1,920×1,080 pixels
`streamcluster`	4096 points, 32 dim	16384 points, 128 dim
`swaptions`	16 swaptions, 5k sims.	64 swaptions, 20K sims.
Rodinia 1.0		
`backprop`	—	65536 input nodes
`bfs`	65536 nodes	1M nodes
`cfd`	fvcorr.domn.097K	fvcorr.domn.193K
`heartwall`	—	609×590 pixels/frame
`hotspot`	512×512 data points	1024×1024 data points
`leukocyte`	—	219×640 pixels/frame
`lud`	256×256 data points	512×512 data points
`needlemanwunsch`	1024×1024 data points	2048×2048 data points
`srad`	512×512 points	2048×2048 points

Table 2. Benchmarks considered in this study along with their input sets.

4 Results

We now present the results obtained by applying the proposed performance analysis methodology. This is done in a number of steps. We first present cycle stacks and some illustrative case studies using cycle stacks to analyze parallel performance bottlenecks. Subsequently, we use the methodology to compare three prevalent benchmark suites, namely SPLASH-2, PARSEC and Rodinia, and we obtain some interesting insights into the (dis)similarity of these benchmark suites with respect to each other. Finally, we study how workload behavior scales with the number of cores and increasing input data set sizes.

4.1 Cycle stacks

Using cycle stacks, software researchers and developers can easily identify the performance bottleneck of an application on a particular platform and study how application behavior changes with varying hardware configurations, while computer architects can use cycle stacks to help optimize architectures.

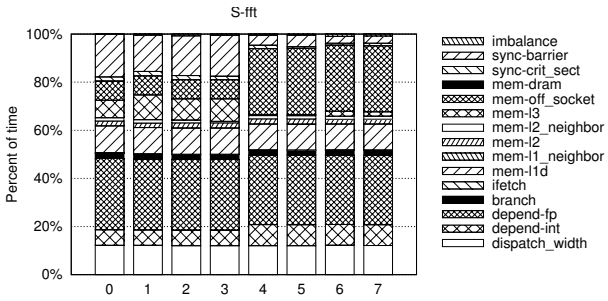

Figure 3. Normalized cycle stacks for each core executing the `fft` benchmark when running the small input set on eight cores.

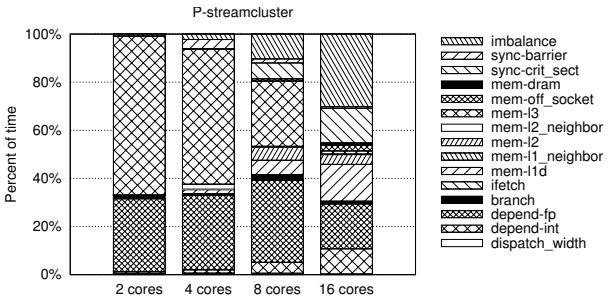

Figure 4. Normalized cycle stacks for the `streamcluster` benchmark with the large input set over core counts from 2 to 16 cores.

Each cycle component represents the amount of time that can be attributed to that type of event. For example, the memory hierarchy components, starting with *mem-*, indicate that the cache miss was resolved by that particular component. In order words, the contribution indicated by *mem-l2* corresponds to the amount of time spent waiting for data that resulted in an L2 hit. In contrast to a pure L2 hit rate, which can be obtained from performance counters or cache-only simulation, the *mem-l2* cycle component shows both the number of L2 hits and their effect on performance. This performance effect in turn depends on the L2 cache access latency and on the amount of overlap that the out-of-order core can provide between the cache access and other, independent instructions. Additionally, time attributed to the *mem-l1_neighbor* component means that the data requested was found in the neighboring L1 on the local socket. The *mem-off_socket* component denotes memory accesses that miss in the local L3 cache but hit in the cache of another processor chip; while *mem-dram* also goes off-chip but needs to access DRAM memory. The *ifetch* and *branch* components result from instruction cache misses and branch misses, respectively. *Dispatch_width* indicates the amount of time that the processor was able to dispatch the full four instructions in a single cycle, and the *depend-int* and *depend-fp* represent the amount of time that the cause of a processor stall was the result of an integer or floating-point instruction dependency. The *sync-barrier* and *sync-crit_sect* are the amount of time the applications spend in barrier or critical section synchronization. Finally, the *imbalance* portion of the stack represents the amount of time that is wasted because of a thread starting late in the execution, or ending early before the last thread exits.

We now present several case studies to demonstrate the utility of cycle stacks for analyzing parallel performance. The first example shows how cycle stacks can identify inhomogeneous behavior among ostensibly homogeneous threads. Figure 3 shows the normalized cycle stacks for all cores in an eight-core simulation of the `fft` benchmark (SPLASH-2) with the small input set size. Although `fft` is a homogeneous benchmark where all threads execute the same code, performance can be inhomogeneous due to memory access behavior. A clear difference among cores' behavior is in the amount of L3 hits versus off-chip memory accesses. Cores 0–3, which are located on the first socket, spend some time waiting for their local L3 cache but perform only a limited amount of off-chip accesses. In contrast, cores 4–7, which are all on the second processor chip, have to get a much larger fraction of their data off-chip (in the L3 cache of the first processor). These off-chip accesses take longer, which causes cores 4–7 to execute more slowly than cores 0–3, which in turn spend the extra time waiting in *sync_barrier*.

The second example shows how cycle stacks can identify how bottlenecks change as the number of cores changes. Figure 4 shows an example of the (normalized) cycle stacks for `streamcluster` (PARSEC) when scaling the system from 2 to 16 cores. When `streamcluster` executes on two cores, most of the performance bottleneck concentrates on the shared L3 cache and floating-point units. As we increase the number of cores, the contributions of individual cycle components vary significantly. From one to eight cores, a super-linear speedup can be seen because of the increase in available cache — this is confirmed in the CPI stacks by the reduction of the *mem-l2* and *mem-l3* components. When running `streamcluster` on 16 cores, even though even more cache is now available and the *mem-l3* component disappears completely, load imbalance and synchronization now contribute most to the overall run time. This shows that in order to improve scaling performance of the `streamcluster` benchmark to higher core counts, load balance and synchronization overheads through critical sections need to be improved, for instance by using work stealing and finer-grained locking.

The third example shows how cycle stacks can identify how bottlenecks change with input (as well as core count). Figure 5 shows cycle stacks for the `srad` benchmark (Ro-

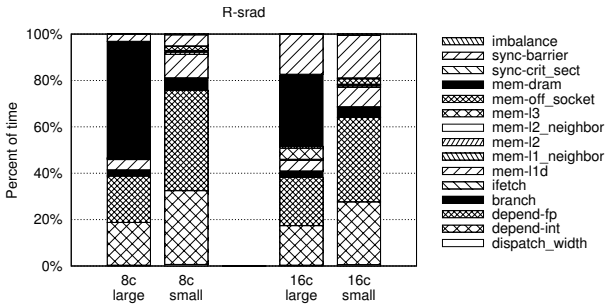

Figure 5. Normalized cycle stacks for the `srad` benchmark with 8 versus 16 threads, and small versus large input.

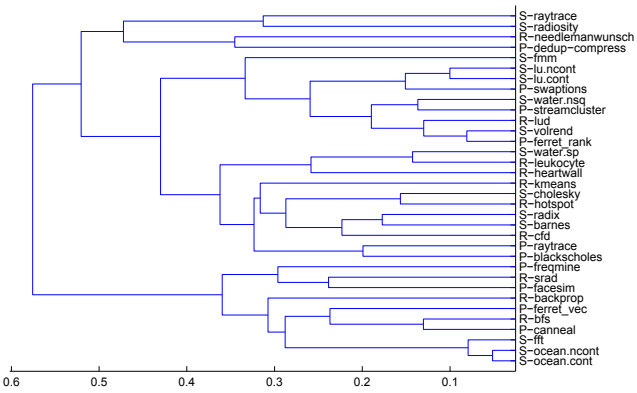

Figure 6. Dendrogram obtained cluster analysis when applied to all benchmarks, assuming 16 threads and large input sets.

dinia) for 8 and 16 cores, and both small and large inputs. As in the preceding example, the cycle stacks show different application behaviors as a function of core count, but also at a given core count, they show different behavior as a function of input size. When using a small input, most of the working set of `srad` fits into the last-level cache, and the time spent on compute units contributes (relatively) more to the total run time. On the other hand, with a large input, `srad` stresses the memory hierarchy which results in a significant fraction of time spent on cache misses and off-chip DRAM accesses.

4.2 Comparing benchmark suites

Now that we are able to compute cycle stacks for all of the benchmarks in the SPLASH-2, PARSEC and Rodinia benchmark suites, we can analyze how different these benchmarks are with respect to each other. Figure 6 shows the overlap of the three benchmark suites in the hierarchical clustering tree. The results are shown for a 16-core machine

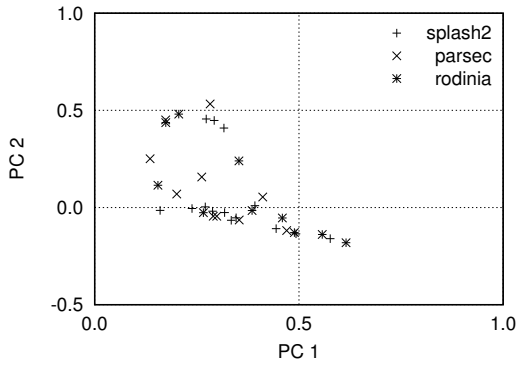

Figure 7. PCA analysis of all benchmarks, assuming 16 threads and large input sets.

with large inputs. We consider six principal components capable of representing over 90% of the total variance. In the figure, the applications grouped in one cluster are more similar than the applications in other clusters. Note that the horizontal axis represents the linkage distance; the distance between two clusters is defined as the average distance between the cluster members. This analysis shows that the three benchmark suites cover a similar application space.

To better understand the first-order cycle components which most affect the differences, Figure 7 plots all applications in a 2-D PCA space with PC1 and PC2. The combination of the two PCs explains 72% of the total variance. As shown in Table 5, floating-point operations (*depend-fp*) contribute most to PC1, while DRAM accesses (*mem-dram*) play the largest role for PC2. This means that a workload with a high value along PC1 exhibits a high fraction of floating-point instructions along the critical path; likewise, a workload with a high value along PC2 is more memory-intensive. Again we can see the three benchmark suites to occupy a similar space. Table 4 adds the other principal components as well. In the table, for each benchmark suite, we report the average and standard deviation for each component across all benchmarks in the suite. PC3 has its main contribution from *depend-int* but also has a high contribution from *branch*, so it is highly correlated with the complexity of control flow in the benchmark. Most of the Rodinia benchmarks have a low value for this component, showing that this suite — compared to SPLASH-2 and PARSEC — contains more regular applications that are purely compute-bound. This is confirmed in Table 3 which reports the benchmarks with the five highest and lowest value for each of the components. All of the bottom five for PC3 are benchmarks from the Rodinia suite.

PC 1 (*depend-fp*)		PC 2 (*mem-dram*)		PC 3 (*depend-int*)		PC 4 (*mem-l3*)		PC 5 (*sync-barrier*)	
R-heartwall	0.62	P-ferret_vec	0.53	P-dedup_Compress	0.34	R-backprop	0.24	P-swaptions	0.32
S-radix	0.58	R-backprop	0.48	P-ferret_rank	0.27	S-lu.ncont	0.16	S-lu.cont	0.32
R-cfd	0.56	S-fft	0.46	S-lu.cont	0.25	R-kmeans	0.15	S-lu.ncont	0.31
S-barnes	0.49	P-canneal	0.45	R-lud	0.24	P-facesim	0.15	R-heartwall	0.29
R-leukocyte	0.49	S-ocean.ncont	0.45	S-volrend	0.24	S-lu.cont	0.10	R-leukocyte	0.25
...		...		...		...		...	
R-bfs	0.17	R-leukocyte	-0.13	R-backprop	-0.07	R-hotspot	-0.06	R-bfs	-0.05
P-canneal	0.17	S-barnes	-0.14	R-kmeans	-0.07	S-fft	-0.07	R-needlemanwunsch	-0.10
S-radiosity	0.16	R-cfd	-0.14	R-heartwall	-0.08	S-ocean.cont	-0.07	P-raytrace	-0.11
R-needlemanwunsch	0.15	S-radix	-0.16	R-hotspot	-0.14	S-radix	-0.09	P-blackscholes	-0.15
P-freqmine	0.14	R-heartwall	-0.18	R-cfd	-0.20	R-heartwall	-0.09	P-dedup_Compress	-0.24

Table 3. PCA analysis of all benchmarks, assuming 16 threads and large input sets. For each PCA component, the main contributor is shown, in addition to the five benchmarks that have the highest and lowest value for this component.

Suite	PC 1 (*depend-fp*)			PC 2 (*mem-dram*)			PC 3 (*depend-int*)			PC 4 (*mem-l3*)			PC 5 (*sync-barrier*)		
SPLASH-2	0.34	±	0.10	0.05	±	0.21	0.09	±	0.09	0.01	±	0.07	0.12	±	0.11
PARSEC	0.29	±	0.10	0.12	±	0.21	0.13	±	0.12	0.04	±	0.05	0.05	±	0.18
Rodinia	0.37	±	0.15	0.07	±	0.23	0.00	±	0.13	0.04	±	0.10	0.07	±	0.12

Table 4. Average and standard deviation of PCA components for each benchmark suite.

CPI component	PC 1	PC 2	PC 3	PC 4	PC 5	PC 6
Cum. sum variance	51.16	71.59	79.73	86.39	91.91	94.24
sync-crit_sect	0.03	—	0.08	0.03	0.13	0.19
sync-barrier	0.19	—	0.42	0.20	0.84	-0.02
mem-dram	0.24	0.94	-0.02	-0.23	—	—
mem-off_socket	0.03	0.03	0.05	0.03	0.08	0.04
mem-l3	0.15	0.17	-0.39	0.89	-0.06	—
mem-l2_neighbor	—	—	—	0.01	—	0.01
mem-l2	0.06	—	0.14	0.08	-0.06	-0.32
mem-l1_neighbor	—	—	—	—	0.01	—
mem-l1d	0.20	-0.02	0.35	0.14	-0.20	-0.28
branch	0.16	-0.02	0.43	0.12	-0.31	0.65
depend-fp	0.85	-0.29	-0.35	-0.24	0.04	0.02
depend-int	0.28	—	0.46	0.11	-0.36	-0.36
dispatch_width	0.12	0.01	0.06	0.07	-0.01	0.48

Table 5. PCA weights.[2]

4.3 Scaling behavior

So far, we considered a single data point per benchmark, i.e., one input (the large input) and one system configuration (16 cores) for each benchmark. This and the next section use PCA analysis to gain insight in how workload behavior changes with varying system parameters and input data set sizes, respectively. In fact, cycle stacks along with PCA analysis allow for a unique characterization to quickly gain insight in how workload behavior changes with varying system and input settings.

Figure 8 shows the PCA plots for both Rodinia and SPLASH-2 as we scale from 4 to 16 threads, assuming large input data sets. The dotted lines in these graphs connect the 4-thread data points with the 16-thread data points; the crosses denotes 16 threads, and the other ends of the dotted lines denote 4 threads. Interestingly, we observe different scaling behavior across benchmarks and benchmark suites. The SPLASH-2 benchmarks mostly scale such that they become less floating-point intensive (scaling towards lower values along PC1) as we increase core counts; however, scaling does not affect memory performance significantly (minor impact along PC2). This suggests that while computation is being distributed across the threads as we increase the number of threads, overheads related to data sharing and communication start to impact performance. This is confirmed by these benchmark's increase along the PC4 axis (not shown), for which *mem-l3* is the main contributor. Rodinia on the other hand, which is more memory-intensive in the 4-threads case compared to SPLASH-2, becomes less memory-intensive with increasing core counts (scaling towards lower values along PC2). This suggests that these applications benefit from the increased cache size that comes with the extra cores, reducing the time spent waiting for DRAM operations. The one exception is the `srad` benchmark which, like SPLASH-2, becomes less floating-point intensive, as we have observed in Figure 5.

[2]Note that hyphens indicate that for the component c, $|c| < 0.01$.

4.4 Input data sets

We now study how input data sets (small versus large) affect workload behavior. In Figure 9, we can see two classes of applications. First, is the class of applications that does not show significant changes when moving between input sizes. When one sees characteristics like these, one can say that for this hardware, these input sizes do not play a significant role in determining the characteristics of the application.

The other class of applications are more input-dependent, meaning that runs with these smaller input sizes

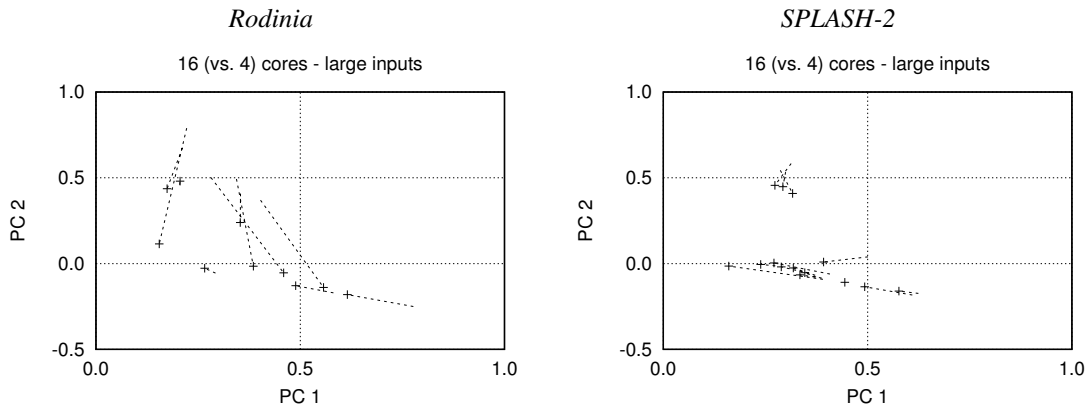

Figure 8. PCA analysis of Rodinia (left graph) and SPLASH-2 (right graph) when scaling from 4 to 16 threads, assuming large input sets. The crosses denote 16 threads, and the other end of the dotted lines denote 4 threads.

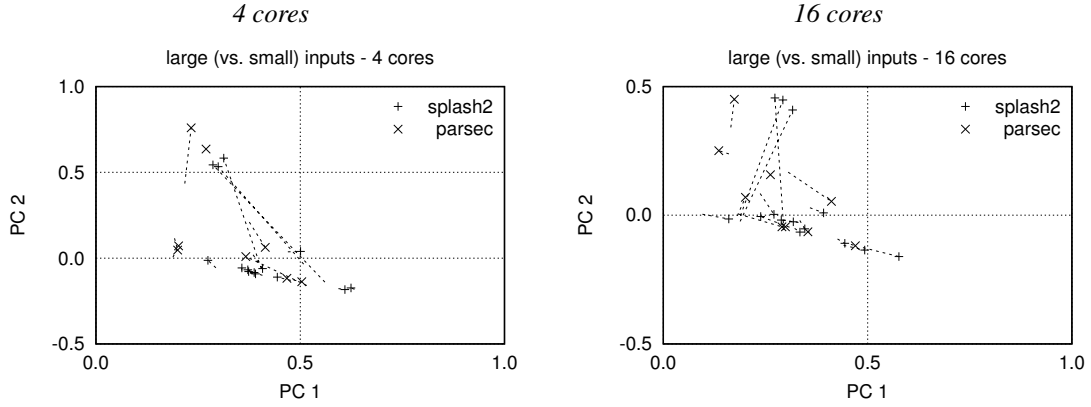

Figure 9. PCA analysis when changing the input set size from small to large; we assume 4 cores in the left graph and 16 cores in the right graph. The crosses denote large input sets, and the other end of the dotted lines denote small input sets.

cannot act as a proxy for performance for these applications. Moreover, the input dependence is not the same for 4-core versus 16-core simulations: although in both cases the memory (PC2) component increases for the large input sets, decreasing the input size generally increases the floating-point (PC1) component for 4-core simulations whereas the 16-core simulations show a decrease.

The fact that small inputs are not representative for large inputs for some workloads has an important implication for architectural simulation of multicore systems. For these workloads, one needs large input sizes which are time-consuming to simulate given the slow speed of contemporary detailed simulators. This observation makes the case for simulation methods that scale with increasing core counts and input data set sizes.

5 Related work

We identify three areas of related work in performance analysis tools for parallel workloads, building cycle stacks, and workload characterization.

5.1 Parallel workload analysis tools

There exist a number of tools for analyzing parallel performance. Examples are Intel's VTune Amplifier XE [13] and Rogue Wave/Acumem ThreadSpotter [16], which enable the end user to identify performance bottlenecks in parallel workloads and help build software that scales for multicore and manycore processors. Although these tools are powerful, they do not provide cycle stacks. Intel's VTune uses hardware performance counters provided by the

hardware; unfortunately, existing hardware counters do not provide enough information for computing accurate cycle stacks [8]. Furthermore, an approach that uses hardware counters in existing hardware cannot extrapolate to future hardware. Rogue Wave/Acumem ThreadSpotter samples a running application to capture a fingerprint of its memory access behavior, and provides feedback to the user to address memory performance problems. The information provided by ThreadSpotter is limited to cache miss ratios and similar aggregate event counts, and does not provide detailed cycle stacks. Our methodology uses simulation to build cycle stacks. While simulation is slower than profiling an application as it runs on real hardware, it allows for extrapolating beyond existing hardware.

5.2 Cycle stacks

Eyerman et al. [8] propose a cycle accounting architecture for constructing CPI stacks on an out-of-order processor core. The key challenge in constructing CPI stacks for out-of-order processors is how to deal with overlap effects so that cycles during which useful work is performed underneath a miss event are accounted as useful work and not as lost cycles, the impact of overlapping miss events (e.g., overlapping memory accesses) are not be double-counted, etc. This paper contributes over the work of Eyerman et al. by proposing a mechanism for quantifying the impact of ILP on the base cycle component; see Section 2.2. Furthermore, we make the case for cycle stacks for analyzing parallel performance, whereas Eyerman et al. focus on single-core processors only. Additionally, the novel hardware performance counters proposed by Eyerman et al. can be used as a building block to perform the analysis described in this paper in real hardware, removing the need for simulation.

Fields et al. [10] propose the notion of *interaction cost* to quantify how two or more events interact with each other during execution on a core. The interaction can be positive (parallel execution), negative (serial execution) or zero (independent or no interaction). They use interaction cost to build dependency graphs that highlight the critical execution path of the execution of a workload on a processor. The critical path analysis proposed in Section 2.2 is similar in concept, except that we characterize ILP only, in contrast to Fields et al. who characterize the critical path throughout the entire processor pipeline. Also, we build cycle stacks for multi-threaded workloads on multicore processors, whereas Fields et al. target single-threaded workloads on single-core processors.

5.3 Workload analysis

Eeckhout et al. [7] propose a workload characterization methodology based on principal component analysis. Their motivation was to understand behavioral differences across workloads in an insightful way. Nevertheless, their work focuses on single-threaded workloads. More recent work by Bienia et al. [2] and Che et al. [5] use this methodology to characterize parallel workloads, namely PARSEC and Rodinia, respectively. However, the workload characterization is limited to some high-level workload characteristics such as instruction mix, working set size and sharing behavior — too high a level to study performance scaling behavior. This paper on the other hand uses cycle stacks as input to PCA to gain insight into application scaling behavior.

6 Conclusion

This paper proposed a methodology that uses cycle stacks and statistical data analysis for analyzing parallel workload performance. Cycle stacks break up total execution time into cycle components that quantify where the cycles have gone, and are measured through simulation using a novel critical path cycle accounting mechanism. Statistical data analysis using principal component analysis (PCA) allows for analyzing general performance trends across workloads and system settings. The paper's main contributions are (i) the evidence for using cycle stacks to analyze parallel workload performance, along with (ii) a case study analyzing and comparing scaling behavior across three prevalent benchmark suites, SPLASH-2, PARSEC and Rodinia. We presented several case studies illustrating the value of cycle stacks for identifying performance scaling bottlenecks in real workloads, due to load imbalance, synchronization, poor memory performance, etc. Using the proposed methodology, we derived the insight that although SPLASH-2, PARSEC and Rodinia stress similar components of the system, their scaling behavior to larger core counts and larger input sets differs: for instance, when increasing the core count, many Rodinia benchmarks are able to better use the extra cache available, whereas several SPLASH-2 applications suffer from increased non-local memory accesses. Our analysis therefore shows that comparing benchmarks and benchmark suites is sensitive to input size and to the machine configuration (number of cores, cache size, etc.). The results also suggest directions in which each suite might fruitfully be expanded to encompass a wider range of scaling behaviors.

Acknowledgments

We thank the reviewers for their constructive and insightful feedback. Wim Heirman and Trevor Carlson are supported by the ExaScience Lab, which is supported by Intel and the Flemish agency for Innovation by Science and Technology (IWT). Additional support is provided by the FWO projects G.0255.08 and G.0179.10, the UGent-BOF

projects 01J14407 and 01Z04109, the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295, and US NSF grant no. CNS-0916908 (ARRA).

References

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, Oct. 2008.

[2] C. Bienia and K. Li. Fidelity and scaling of the PARSEC benchmark inputs. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, Dec. 2010.

[3] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *To appear at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct. 2009.

[5] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, Dec. 2010.

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

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

[8] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–184, Oct. 2006.

[9] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems (TOCS)*, 27(2):42–53, May 2009.

[10] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn. Interaction cost and shotgun profiling. *ACM Trans. Archit. Code Optim.*, 1(3):272–304, Sept. 2004.

[11] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 307–318, Feb. 2010.

[12] Intel Many Integrated Core Architecture. `http://www.intel.com/pressroom/archive/releases/2010/20100531comp.htm`

[13] Intel VTune™ Amplifier XE 2011. `http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/`

[14] R. A. Johnson and D. W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, fifth edition, 2002.

[15] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, Jan. 2010.

[16] Rogue Wave Acumem ThreadSpotter™. `http://www.roguewave.com/products/threadspotter.aspx`

[17] V. Uzelac and A. Milenkovic. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 207–217, 2009.

[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture (ISCA)*, pages 24–36, June 1995.