Working Set Model for Multithreaded Programs

Kishore Kumar Pusukuri
Oracle Inc., Santa Clara, CA, USA.

Abstract
Knowledge of the working set of pages associated with an application provides an opportunity for effective allocation of resources in multicore multiprocessor systems. Various techniques for approximating the working set size using either simulations or program traces have been proposed. However, these techniques are very expensive, and therefore not practical for *dynamically* optimizing resources. To alleviate this problem, in this work, we develop a statistical model based on machine learning techniques for approximating the working set size of multithreaded programs running on multicore multiprocessor systems. The basic idea is to correlate the working set size of a program with its resource usage characteristics, such as resident set size and TLB miss rate. Through extensive experimentation with 20 multithreaded programs from SPEC OMP2012 and PARSEC on a SPARC T4-4 running Oracle Solaris 11™, we demonstrate that the model has 96% prediction accuracy and its overhead is negligible.

1 Introduction

Multicore multiprocessor systems with large caches and many cores have become ubiquitous. Large caches used in shared memory multicore architectures can avoid high memory access time only if data is referenced within the address scope of the cache. Therefore, locality is a key issue in achieving high performance for multithreaded applications running on multicore systems [1, 19]. Locality (temporal locality) is often expressed in terms of working sets [11]. The working set of an application is a measure of its dynamic memory demand in an execution window [10, 23]. The working set size (WSS) of an application is computed by multiplying the number of pages referenced *over an interval* with the size of the page. WSS has been used to model resource sharing among concurrent tasks of an application [10] and for effective resource management in the Cloud [24]. Therefore, knowing the WSS of an application provides an opportunity for effective allocation of resources and thus improved performance.

Various techniques for approximating WSS using either simulations or program traces have been proposed [2, 4, 7, 10, 13, 23]. Although using these techniques we can determine the WSS of a *particular* program, the overhead of these techniques is very high and therefore not appropriate for *dynamically* optimizing resources. Moreover, most of the above existing techniques were evaluated using single-threaded programs on machines with very few cores. Oracle Solaris 11 provides the number of pages referenced by an application running on SPARC systems through the *proc* file system (/proc/pid/pagedata). However, it is very expensive to derive the WSS through the proc file system, as the overhead increases with the size of the working set. For example, the overhead of measuring the working set sizes of 20 multithreaded programs from PARSEC [3] and SPEC OMP2012 [25] on SPARC T4-4, ranges from 80 millisecs to 150,000 millisecs.

To address the above problems, in this paper, we develop a statistical model based on supervised learning for approximating the WSS of a *multithreaded program* running on a *multicore multiprocessor system*. To develop this model, first, we study the important resource usage characteristics of 20 emerging multithreaded programs from the PARSEC and SPEC OMP2012 benchmark suites on a SPARC T4-4 running Oracle Solaris 11. The basic idea is to correlate the WSS of an application with its important resource usage characteristics, such as resident set size (RSS) and TLB miss rate. We collect these characteristics using performance monitoring counters and simple utilities available in modern operating systems. Through extensive experimentation, we demonstrate that the model has 96% prediction accuracy. Moreover, the overhead of this model is negligible. Thus, we can use this model dynamically for effective allocation of resources in multicore multiprocessor systems. We also briefly present two potential applications of the above model to maximize the performance of multithreaded programs running on multicore multiprocessor systems.

PARSEC:	bodytrack (BT), fluidanimate (FA), ferret (FR), facesim (FS), streamcluster (SC), swaptions (SW);
SPEC OMP2012:	applu (AL), botsalgn (BA), bt331 (BB), botsspar (BS),bwaves (BW), fma3d (FM), ildbc (IL), imagick (IG), kdtree (KD), md (MD), mgrid (MG), nab (NM), smithwa (ST), swim (SM)

Table 1: The 20 multithreaded programs and their short names. These programs mainly stress CPU and main memory.

The key contributions of our work are as follows:

- We identify that *resident set size, TLB miss rate, number of threads, and LLC miss rate* are strongly correlate with the working set characteristics of multithreaded programs running on multicore multiprocessor systems.
- We demonstrate the usage of machine learning techniques in developing *simple and robust* models for characterizing working set sizes of emerging multithreaded applications.

The remainder of this paper is organized as follows. Section 2 presents the development of statistical models for approximating working set size in detail. We briefly describe our future work in Section 3. Finally, related work and conclusions are given in Sections 4 and 5.

2 Characterizing Working Set Size

The working set of pages associated with an application is the collection of its most recently used pages over a time interval [10]. The working set size (WSS) is computed by multiplying the page-size with the number of pages referenced over a time interval. Characterizing the WSS is a challenging problem as it significantly varies from application to application. For example, a well-tuned scientific application can have a small working set of a few kilobytes, while a large database application can have a huge working set running into several gigabytes. Moreover, several factors affect the WSS of a multithreaded application running on multicore multiprocessor systems.

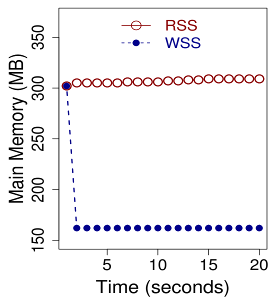
The basic idea for developing a model for approximating WSS is correlate it with important resource usage characteristics (i.e., factors) of the application. Therefore, identifying important factors is the key in developing an effective model to characterize the WSS of multithreaded applications. We use statistical models to capture important factors for approximating WSS. These models are constructed using supervised learning, where a set of input-output values is first observed and then a statistical model is trained to predict similar output values when similar input values are observed [15].

2.1 Identifying Important Factors

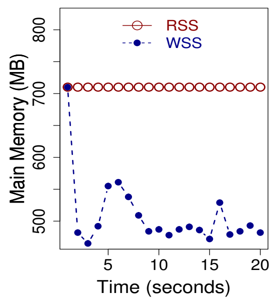
To identify important resource usage characteristics that affect WSS, first we study 20 multithreaded programs from the PARSEC and SPEC OMP2012 with varying number of threads on a SPARC T4-4 running Oracle Solaris 11. These programs mainly stress CPU and main memory. Table 1 lists the chosen programs. The implementations of the PARSEC programs are based upon *pthreads* and

SPARC T4-4:	4 $\times$ SPARC T4 Processors @3.0 GHz;
	Total 256 vCPUs (i.e., 4 $\times$ 64 vCPUS);
	L3: 4 MB; RAM: 512 GB; OS: Oracle Solaris 11$^{\text{TM}}$

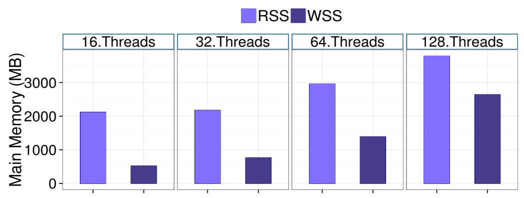
Table 2: The description of SPARC T4-4.



(a) Facesim.



(b) Fluidanimate.



(c) Imagick.

Figure 1: Although RSS is correlated with WSS, *it is not alone* enough to accurately approximate WSS. The number of threads used to run the multithreaded program affects its WSS and RSS.

we ran them using native inputs (i.e., the largest inputs available). The SPEC OMP2012 programs were run on medium sized inputs. The range of the execution times of the programs is from 50 seconds to 4800 seconds.

Table 2 provides the description of SPARC T4-4: it has four SPARC T4 processors, each of the processors (or sockets) has 64 virtual CPUs (vCPUs), and a total of 256 vCPUs. Since most of the above 20 programs exhibit poor performance because of high lock contention when running them with 256 threads on our SPARC T4-4, we assume that the above mentioned programs run with a maximum of 128 threads on SPARC T4-4. Therefore, we run the programs with 16, 32, 64, and 128 threads and collect their important resource usage characteristics for developing a statistical model to approximate WSS.

Resident Set Size & Number of Threads. The resident set size (RSS) is a measure of how much physical memory is actually being used by the application [13]. Therefore, RSS is correlated with the WSS. However, it is not sufficient to *accurately approximate* the WSS of multithreaded programs (see Figure 1 (b)). Figure 1 shows the RSS and the WSS of three multithreaded programs: facesim (FS), fluidanimate (FA) from PARSEC, and Imagick (IG) from SPEC OMP2012. We ran FS and FA with 64 threads and IG with varying number of threads (16, 32, 64, and 128). The utility ps(1) is used to measure *resident set size (RSS)*. As Figure 1 shows, while the WSS of FS is steady, the WSS of FA varies. As we can see in Figures 1 (a), (b), and (c):

- RSS is not sufficient to accurately approximate WSS.
- Depending on the application WSS varies significantly. Figure 1 (b) shows this.
- The number of threads used to run the multithreaded program significantly affects both WSS and RSS. We can see this in Figure 1 (c).

Therefore, both RSS and number of threads are important factors for approximating WSS.

TLB Miss Rate & Number of Threads. Translation lookaside buffer (TLB) is used to improve the speed of virtual to physical address translation in modern computer systems. The number of entries in a TLB multiplied by the page size is defined as TLB reach. The TLB reach is critical to the performance of an application. If the TLB reach is not enough to cover the working set of a process, the process may spend a significant portion of its time satisfying TLB misses [10]. Therefore, TLB miss rate is one of the important application characteristics for approximating WSS.

We also observe that multithreaded programs based on *data parallelism* experience smaller TLB miss rate when running them with more threads compared to running the same program with relatively small number of threads (or sequential version) on multicore multiprocessor systems. This is because, when we run the program with multiple threads, the thread scheduler migrates threads across all the cores (and sockets) of the system for balancing load. This allows threads to share the TLB cache of the whole system and also the working set is shared among all the threads. Therefore, the number of threads used to run a multithreaded program significantly affects TLB miss rate and thus it is one of the important factors for approximating WSS.

LLC Miss Rate & Thread Migrations. The thread schedulers dynamically distribute threads across multiple processors to balance the load across the processors. However, the distribution of threads across processors of a multicore multiprocessor system impacts important application resource usage characteristics such as data locality, last-level cache misses (LLC misses) and thus performance of the multithreaded application. Typically, programs with large working set sizes exhibit high LLC miss rates.

Predictor	Description
rss	average resident set size.
tlbpi	average TLB misses per instruction.
mpi	average LLC misses per instruction.
tlbps	average TLB misses per second.
mps	average LLC misses per second.
load	threads to virtual CPUs ratio i.e., (#threads / 256).

Table 3: The initial six predictors of WSS (the response variable) of a multithreaded application running on a multicore multiprocessor system. We express WSS in terms of MB.

Therefore, we consider RSS, TLB miss rate, number of threads used to run the application, and LLC miss rate as potential predictors (or important resource usage characteristics) of WSS. These predictors are used as inputs to the statistical models. Table 3 lists the predictors.

2.2 Data Collection

We run the above 20 multithreaded programs (Table 1) with varying number of threads (16, 32, 64 , and 128) and collect input data (values of the predictors) for developing the models. From the runs of the 20 programs, we collect 80 data points, where each data point is a 7-tuple containing the 6 predictors (shown in Table 3) and the observed value of *WSS* as the target parameter. Here each of the 20 programs contributes 4 data points. We use cputrack(1) utility [18] (accessing performance event counters) for collecting *TLB misses per instruction (tlbpi), LLC misses per instruction (mpi), TLB misses per second (tlbps), and LLC misses per second (mps)*. While ps(1) utility is used to measure *resident set size (RSS)*, *WSS* data is derived by collecting the number of pages referenced information through the proc file system.

Collecting WSS Data. Oracle Solaris 11 on SPARC provides the information of referenced pages of a process through the proc file system. Therefore, in this work, WSS data is derived by collecting the number of pages referenced information through the `/proc/pid/pagedata`. The proc pagedata corresponding to a process is another representation of the process's address space and provides page-level reference and modification tracking [18]. The proc pagedata file contains the information of total pages mapped and how many pages are referenced or modified. Therefore, accessing the proc pagedata file enables tracking of address space references and modifications on a per-page basis. However, depending on the number of pages mapped, this takes a huge amount of time -- from 80 milliseconds to 150,000 milliseconds for the programs we used in this work. Thats why it is not appropriate to collect WSS data through the proc file system for dynamically optimizing resources. The range of WSS values of the 20 programs is from 4MB to 21,000 MB. We express WSS in terms of MB in this work.

2.3 Finding Important Predictors

To balance the prediction accuracy and the cost of approximation, we use forward and backward input selection techniques with Akaike Information Criterion (AIC) [15] for finding important predictors among the six initial predictors. The AIC is a measure of the relative goodness of fit of a statistical model. AIC deals with the trade-off between the complexity of the model and the goodness of fit of the model. The R stepAIC() [26] function uses the AIC criterion for weighing the choices, which takes proper account of the number of parameters fit. Therefore, using R stepAIC() method, we identified the two most important predictors: *rss* and *tlbpi*. Although number of threads and LLC miss rate are other important factors for approximating WSS, TLB miss rate includes the affect of these on WSS. That's why RSS and TLB miss rates are enough to approximate WSS of multithreaded programs.

Moreover, we also tested the predictors for the multicollinearity problem to develop robust models. Multicollinearity is a statistical phenomenon in which two or more predictor variables in a multiple regression model are highly correlated. In this situation the coefficient estimates may change erratically in response to small changes in the model or the data. We use *R Variance Inflation Factor (VIF)* method to observe the correlation strength among the predictors [20]. If $VIF > 5$, then the variables are highly correlated [26]. The VIF values of the predictors *rss* and *tlbpi* are around 1.04. Therefore, there is no multicollinearity problem. In other words, the model developed using these two predictors does not overfit the training data and has a better expected prediction accuracy on test data.

2.4 Developing Models

Using the above two important predictors (rss and tlbpi) we developed three popular models based on supervised learning techniques. The models are: 1) Linear Regression (LR); 2) Regression Tree (RT); and 3) K Nearest Neighbour (KNN) [15, 16].

2.4.1 Linear Regression (LR)

Linear regression models are simple and often provide an adequate and interpretable description of how the inputs affect the output. Linear regression models the relationship between a scalar dependent variable y and one or more explanatory variables denoted X. Given a vector of inputs $X^T = (X1, X2, ..., Xp)$, we predict the output Y via the model [15] shown in Equation (1). The term β_0 is the intercept and β_j represents the vector of coefficients. Equation (2) shows the linear regression model developed with the above data set. We use R lm() [26] to develop the model.

$$\mathsf{Y} = \beta_0 + \sum_{j=1}^{P} \mathsf{X_j} \beta_j \tag{1}$$

$$\mathsf{WSS} = (-150) + (1.015 * \mathsf{rss}) + (-23830 * \mathsf{tlbpi}) \tag{2}$$

2.4.2 Regression Tree (RT)

Recursive binary partitioning is a popular tool for regression analysis. Conditional inference trees estimate a regression relationship by binary recursive partitioning in a conditional inference framework. Roughly, the algorithm works as follows: 1) Test the global null hypothesis of independence between any of the input variables and the response. Stop if this hypothesis cannot be rejected. Otherwise, select the input variable with the strongest association to the response. This association is measured by a p-value corresponding to a test for the partial null hypothesis of a single input variable and the response. 2) Implement a binary split in the selected input variable. 3) Recursively repeat steps 1) and 2) [16].

We use R ctree() [26] to develop the model. The internal node of regression tree represents test on an attribute, each branch represents outcome of test and each leaf node represents the number of data points (n) fall into it and the response value (y). Regression trees are known to be robust to the effect of outliers. Moreover, they are simple to understand and interpret.

2.4.3 K Nearest Neighbor (KNN)

Despite its simplicity, KNN has been successful in many classification problems and regression problems, including handwritten digits, satellite image scenes, etc. Nearest-neighbor methods use those observations in the training set T closest in input space to x to form Y. Specifically, the KNN fit for Y is defined in Equation (3). Here $N_k(x)$ is the neighborhood of x defined by the k closest points x_i in the training sample. Closeness implies a metric, which in this work is Euclidean distance. In other words, we find the k observations with x_i closest to x in input space, and average their responses [15]. We use R knn.reg() [26] to develop the model. In this work, we use two neighbors (k = 2) as it is giving the best prediction accuracy.

$$\mathsf{Y(x)} = \frac{1}{\mathsf{k}} \sum_{\mathsf{x_i} \in \mathsf{N_k(x)}} \mathsf{y_i} \tag{3}$$

2.5 Model Selection

We use both WEKA [9] and R [26] for selecting the best among the above three models. For this, we evaluate the three models using a 10-fold cross-validation (CV) test [15]. In a 10-fold CV test, we split the data (80 data points) into 10 equal-sized partitions. The function approximator is trained using all the data except for one partition and a prediction is made for that partition. Table 4 shows the normalized root mean squared error (NRMSE) values and the prediction accuracies of the three models. Here, the prediction accuracy is expressed in terms of

Model	NRMSE	Prediction Accuracy (%)
LR	13.8	86.2
DT	7.9	92.1
KNN	4.3	95.7

Table 4: Cross-validation (10 fold) test results.

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}|F_i - A_i|^2} \quad (4)$$

$$NRMSE = \frac{RMSE}{A_{max} - A_{min}} * 100 \quad (5)$$

where A_i is the actual value and F_i is the forecast value.

normalized root mean squared error (NRMSE) [27]. The prediction accuracy (%) is defined as (100 - NRMSE). As we can see in Table 4, although all the three models perform well, KNN model is found to have the highest prediction accuracy among the three models. NRMSE is defined in Equation. 5.

Figure 2 shows the visual summary of how the models perform in terms of *normalized errors* in the 10-fold CV test. As Figure 2 shows the two regions (shaded and non-shaded) represent performance of different models in terms of normalized errors. For example, in Figure 2 (a), while the x-coordinate of each dot represents the normalized error of the LR model for a test data point, the y-coordinate represents the normalized error of the KNN model for the same test data point. The region with more dots reflect lower prediction accuracy by the model. As we can see in Figures 2(a) and 2(b), the KNN model has high prediction accuracy and outperforms both the LR and RT models. Therefore, we chose the KNN model as the working set model.

Algorithm 1 summarizes the methodology used in this work for developing the best model for approximating working set size of a multithreaded application running on a multicore multiprocessor system.

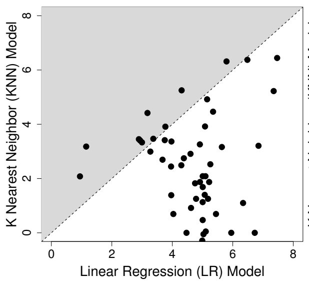
Approximating WSS of SPEC jbb2005 & memcached. While the KNN model achieves 96% prediction accuracy on the above 20 parallel scientific applications, its prediction accuracies for the workloads memcached [17] and SPEC jbb2005 [25] are 93% and 88% respectively. Here the workload based on memcached is the Data Caching Benchmark from the CloudSuite [12].

2.6 Overhead of Working Set Model

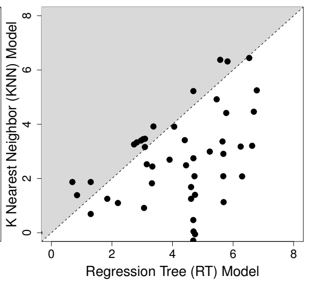
The *overhead* for approximating WSS of a multithreaded program with the working set model (i.e., the KNN model) is 24 microseconds. However, as we described before (Section 2 B), the overhead of measuring WSS through the *proc* file system is very high -- from 80 millisecs to 150,000 millisecs. Therefore, the overhead of the KNN model is negligible.

Algorithm 1: The methodology for developing the best model for approximating WSS.

```
Input: Training data and test data.
Output: The best model for approximating WSS.
1 Develop a linear regression model with all reasonable
  predictors of WSS using Training Data.
2 Select important predictors among the initial
  predictors using Akaike Information Criterion [15].
3 Develop different models using the important
  predictors selected in step 2.
4 Use cross-validation tests (on both Training Data and
  Test Data) to select the best model among the models
  developed in step 3. Here, the best model is the
  model that produces the lowest normalized root mean
  squared error.
```



(a) KNN vs LR.



(b) KNN vs RT.

Figure 2: A visual summary of how the models perform in terms of *normalized errors* in the 10-fold CV test. The two regions (shaded and non-shaded) represents performance of different models in terms of normalized errors. While the x-coordinate of each dot represents the normalized error of a model for a test data point, the y-coordinate represents the normalized error of a different model of the same test data point.

3 Future Work

3.1 Working Set Aware Scheduling

We observe that multithreaded programs with small WSS (e.g., fewer than 16MB on SPARC T4-4) achieve high performance with *grouping threads*, i.e., running the threads on a few processors compared to *spreading threads* uniformly across all the processors (i.e., running with the default Solaris scheduler). However, for the programs with large working sets, spreading yields higher performance than grouping on multicore multiprocessor systems. In our preliminary evaluation, choosing between spreading and grouping based on the WSS of the application significantly improves performance of several multithreaded programs. For example, *bodytrack*, a high lock contention program from the PARSEC benchmark suite achieves a 31% reduction in running time. Therefore, we would like to develop a thread scheduling technique based on the WSS model for effectively scheduling multithreaded programs on multicore multiprocessor systems.

3.2 Resource Management in the Cloud

Virtual machines (VMs) have become one of the primary computing environments in the cloud [24]. One benefit of virtualization is the ability to save and restore the state of a running VM. However, depending on the application, restoring the saved memory image takes significant amount of time. Therefore, memory management is crucial for this restoring process. Since the overhead of the WSS model developed in this work is negligible, it can play a role in allocating memory resources effectively when restoring VMs in the Cloud. Moreover, we can optimize memory resources while deploying multithreaded applications in the cloud by approximating their working set sizes using the above WSS model.

4 Related Work

Several researchers have explored working set characteristics for understanding the resource requirements of the programs [10, 13]. Denning [10] proposes a model for estimating working set sizes to capture the degree of locality in how a process accesses memory. The model affords a convenient way to determine which information is in use by a computation and which is not and therefore makes it easier to determine memory demands [10].

Darryl Gove [13] analyzes the memory demands of the SPEC CPU2006 benchmark programs by measuring their working set sizes using simulator [8]. Darryl[22] evaluates the idea of WSS in relationship to TLB misses for the SPEC CPU2000 benchmark programs. In another context, Cantin and Hill [5] look at the idea of WSS at the level of L1 caches and evaluate the decline in the number of cache misses as the cache size increases. Similarly, Hallnor and Reinhardt [14] investigate the impact of data compression on the WSS of the SPEC CPU2000 benchmark programs. However, the above works only consider single-threaded programs.

The working set model used by the WSClock algorithm [6] considers the number of pages referenced in a time interval as a working set and leverages this information for deciding which page to replace next. However, since this model does not consider page-sizes, it is not possible to compute the working set sizes of programs. Therefore, it is not appropriate for effective scheduling of threads across cores of multicore multisocket system.

Bellosa and Steckermeier [1] use hardware performance counters to detect sharing among threads and to colocate them onto the same processor. They stress the importance of using locality information in thread scheduling for performance scalability of NUMA multiprocessors. Bellosa proposes using TLB information to reduce cache misses across context switches and maximized cache reuse by identifying threads that share the same data regions. The idea is to schedule threads that share regions sequentially one after each other to maximize the chance of cache reuse.

Several other researchers use program traces and simulations for measuring (or approximating) WSS [2, 4, 7, 10, 13, 23]. However, this is very expensive and using memory access traces of a program, we can only measure the working set size of a *particular* program. [24] proposes a technique called *access-bit tracing* for measuring working set size for better resource management in scheduling virtual machines. However, as the authors mention, it has high overhead. Therefore, these techniques are not appropriate for exploiting the WSS information dynamically for effective resource management. Moreover, most of the above existing techniques are evaluated using single-threaded programs on machines with very few cores.

In this work, we develop a simple and robust model using machine learning techniques for approximating the WSS of multithreaded programs running on multicore multiprocessor systems. Unlike the existing techniques, using this model we can approximate the WSS of multithreaded programs with negligible overhead.

5 Conclusions

Working set size (WSS) characteristics of a multithreaded application are vital in developing techniques to optimize resources in both multicore multiprocessor systems and virtualized cloud environments. However, the existing techniques for approximating working set sizes are very expensive and therefore not appropriate for dynamically optimized resources. This work presents a simple model for approximating WSS of multithreaded programs running on multicore multiprocessor systems that incurs negligible overhead and yet achieves 96% accuracy in predicting working set sizes. Finally, we present two potential applications of this model for achieving high performance for multithreaded applications and for optimizing resources in virtualized Cloud environments.

6 Acknowledgements

The author would like to thank Dave Dice, Timothy Farkas, Darryl Gove, Blake Jones, Bart Smaalders, and Darrin Johnson for their help throughout this work. The author would like to also thank Lorenzo Alvisi, for his feedback on earlier drafts of this manuscript.

References

[1] F. Bellosa and M. Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. J. Parallel Distrib. Comput. 37, 1 (August 1996).

[2] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *SIGMETRICS*, 2005.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *PACT*, 2008.

[4] P. Bryant. Predicting Working Set Sizes. In *IBM Journal of Research and Development*. Volume 19 Issue 3, May 1975.

[5] J. F. Cantin and M. D. Hill, Cache performance for selected SPEC CPU2000 benchmarks. In *ACM SIGARCH Computer Architecture News*, v.29 n.4, September 2001.

[6] R.W. Carr and J.L. Hennessy. WSCLOCK -- a simple and effective algorithm for virtual memory management. In *SOSP*, 1981.

[7] A. Chauhan and C. Shei. Static reuse distances for locality-based optimizations in MATLAB. In *ICS*, 2010.

[8] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In *SIGMETRICS*, 1994.

[9] S.J. Cunningham and P. Denize. A tool for model generation and knowledge acquisition. In *Artificial Intelligence and Statistics*, pages 213-222, 1993.

[10] P. J. Denning, The working set model for program behavior. In *Communications of the ACM*, v.11 n.5, p.323-333, May 1968.

[11] Y. Etsion and D. Feitelson. Exploiting Core Working Sets to Filter the L1 Cache with Random Sampling. In *IEEE Transactions of Computers*. 61, 11 (November 2012), 1535-1550.

[12] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ASPLOS*, 2012.

[13] D. Gove. CPU2006 working set size. In *ACM SIGARCH Computer Architecture News archive*, News 35, 1 (March 2007), 90-96.

[14] E. G. Hallnor and S. K. Reinhardt. A Unified Compressed Memory Hierarchy. In *HPCA*, 2005.

[15] T. Hastie, R. Tibshirani, and J. H. Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction. In *Springer Series in Statistics*, 2nd ed. 2009.

[16] T. Hothorn, K. Hornik, and A. Zeileis. Unbiased Recursive Partitioning: A Conditional Inference Framework. In *Journal of Computational and Graphical Statistics*, 15 (3), 651-674. 2006.

[17] F. Itzpatrick. Distributed caching with memcached. Linux Journal 2004, 124 (Aug. 2004), 5.

[18] J. Mauro and R. McDougall. 2006. Solaris Internals (2nd Edition). In *Prentice Hall Publications*, Upper Saddle River, NJ, USA.

[19] A. Mendelson and F. Gabbay. 2001. The effect of seance communication on multiprocessing systems. In *ACM Trans. Comput. Syst*. 19, 2 (May 2001), 252-281.

[20] K. K. Pusukuri, D. Vengerov, and A. Fedorova. A Methodology for developing simple and robust power models. In *WIOSCA*, 2009.

[21] J. R. Rosell and J. Dupuy. The design, implementation, and evaluation of a working set dispatcher. In *Communications of ACM*, Volume 16, Issue 4, 247-253, April 1973.

[22] S. Sair and M. Charney. Memory behavior of the SPEC2000 benchmark suite. *IBM Research Report*, RC 21852, 2000.

[23] X. Xiang, B. Bao, C. Ding, and Y. Gao. Linear-time Modeling of Program Working Set in Shared Cache. In *PACT*, 2011.

[24] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr. 2011. Fast restore of checkpointed memory using working set estimation. In *VEE*, 2011.

[25] SPEC and the benchmark names SPEC jbb2005, SPEC OMP2012 are registered trademarks of the Standard Performance Evaluation Corporation. For more information, see `www.spec.org`.

[26] R stepAIC() lm(), stepAIC(), vif(), ctree(), knn.reg(). `http://www.statmethods.net/`

[27] Normalized Root Mean Squared Error. `http://www.rforge.net/doc/packages/hydroGOF/nrmse.html`