Building Parameterized Models for Black-Box Applications

Gabriel Marin and John Mellor-Crummey
Department of Computer Science
Rice University

1 Introduction

Characterizing and modeling the performance of parallel applications has been a long-standing goal of computer science research. An accurate performance model for an application has many uses including understanding its scalability, providing guidance for resource selection at launch time, guiding on-line performance monitoring and run-time adaptation, and providing input for design of architectures tailored to an application's needs.

Building accurate performance models for parallel applications is difficult. Simply knowing the number of floating-point operations a scientific application executes provides little indication of its performance. Scientific codes rarely achieve peak performance. On a single node, memory hierarchy latency and bandwidth are significant limiting factors. Also, an application's instruction mix can dramatically affect performance; today's superscalar processors can execute multiple instructions in parallel if they are provided with the right mix of instructions. For parallel programs, communication frequency, communication bandwidth and serialization complicate the situation further.

Traditionally, analysts have built models of an application's performance manually. This requires a thorough understanding of the algorithms that are used in an application, as well as their implementation. The advantage of manually constructing such models is that they can be very precise. However, building accurate models in this way is laborious and painstaking.

Our approach aims at building parameterized models of black-box applications in a semi-automatic way, in terms of architecture-neutral Application Signatures. We build models using information from both static and dynamic analysis of an application's binary. We use static analysis to construct the control flow graph of each routine in an application and to look at the instruction mix inside the most frequent executed loops. We use dynamic analysis to collect data about the execution frequency of basic blocks, information about synchronization among processes and reuse distance of memory accesses. By looking at application binaries instead of source code, we are able to build language-independent tools and we can naturally analyze applications that have modules written in different languages or are linked with third party libraries. By analyzing binaries, the tool can also be useful to both application writers and to compiler developers by enabling them to validate the effectiveness of their optimizations.

2 Collecting the Data

To semi-automatically build an architecture-neutral model of a black-box parallel application, we collect several types of dynamic information. At each synchronization point in the program, we record the communication partner and the amount of data sent and/or received. Between synchronization points we record a characterization of the computation expressed as a histogram of basic blocks executed. Separately, we record a characterization of an application's memory access behavior in terms of reuse distance seen by each program reference.

To collect the data, we instrument the binary using a tool based on the EEL library. The instrumenter has two modes. In the first mode it instruments the program to collect communication and computation histograms. In the second mode it instruments the program to collect information about memory reuse distance. The intrumenter analyzes each routine by building its control flow graph, computing Tarjan intervals on the graph (to recover loop nesting structure) and determining the places where instrumentation must be inserted.

We characterize computation by collecting a histogram of basic blocks executed between synchronization points. To capture this information, the instrumenter places instrumentation on selected edges such that the estimated overhead for executing instrumentation code is minimized. At each instrumentation point, a data collection routine, supplied in a shared library by the instrumenter, is invoked to increment a count associated with the corresponding control flow edge. The edge count information is maintained in a sparse data structure. In a post-processing phase following execution we use the counts on instrumented edges to reconstruct counts for all edges and basic blocks.

Using a similar approach, we instrument all memory instructions to collect dynamic information about the memory access behavior. Before each memory reference we invoke a library routine that records a histogram of reuse distance values, where reuse distance is a measure of the number of distinct memory locations touched since the last access to the location being accessed by the current reference.

The instrumentation infrastructure that we developed for measuring this information about program behavior is quite flexible and can be easily adapted to collect other information or to perform different types of online analysis.

3 From Data to Parameterized Models

The data collection infrastructure has stabilized and is largely complete. The principal challenge is to assimilate the data that can be collected with our instrumentation infrastructure into accurate models of application performance.

To build parameterized models that enable us to predict performance for data sizes that we haven't measured, we need to collect data from multiple runs with different and determinable input parameters. Input parameters should include a numerical measure of the problem size and the number of processors used during the execution. While building a model for an entire parallel application is our ultimate aim, we have begun the process of building parameterized models for single node executions. This task alone is a hard problem; we tackle it in several stages.

Using static analysis information for an application, along with dynamic measurements of its execution behavior, the post-processing tool constructs an annotated control flow graph that contains information about loop nesting structure and the execution frequency of each basic block. From this representation, we identify paths in the control flow graph and compute their associated frequencies. Inside nested loops, we work from the inside out; no basic block is considered at more than one loop level. These paths serve as input for an instruction schedule analysis tool that computes an estimate for the execution cost of each path.

To compute the execution cost associated with a path for a (possibly different) target architecture, we translate the instructions present in the basic blocks of the (SPARC) application binary into instances of generic RISC instruction classes. We defined a set of generic RISC instruction classes and a module for translating SPARC binary instructions to generic instructions. We built a configurable scheduler that is initialized with an architecture description that enables us to compute predicted execution times for the specified target architecture. The architecture description defines the number and type of execution units, and the latency, repeat rate and what execution units each generic RISC instruction can be executed on. At this moment the scheduler considers each memory access as a primary cache hit; it does not consider the fetch/decode stages and does not simulate the branch-prediction unit if the processor has one. We are currently working on translating our data on memory reuse distance into an estimation of latency for a given target memory hierarchy. Currently, we use SPARC-based computers from Sun to collect the data and analyze the binaries, we use the scheduling tool to predict performance on a MIPS R12000 processor, and we validate our predictions against actual executions on a MIPS R12000 by collecting performance measurements with hardware counters.

Our plan is to build a model for the execution frequency of each loop, parameterized by the problem size or any other input parameter. Using the execution frequency of multiple runs in which one input parameter is modified and all the others are maintained constant (e.g. varying the problem size), we can compute a curve parameterized by that measure.

We are using this approach to model the behavior of each memory instruction and to predict the fraction of hits and misses for a given problem size and cache configuration. We intend to use this information to improve the prediction of the scheduler by taking into account the contribution of the memory hierarchy. Our first attempt at predicting if a memory instruction is a hit or a miss for a given problem size was to look only at the average reuse distance for that instruction. We were able to produce a very accurate parametric model for the average reuse distance of memory accesses in Sweep3D (see Figure 1), a three-dimensional particle transport problem used as an ASCI benchmark for evaluating high performance parallel architectures. However, this model proved

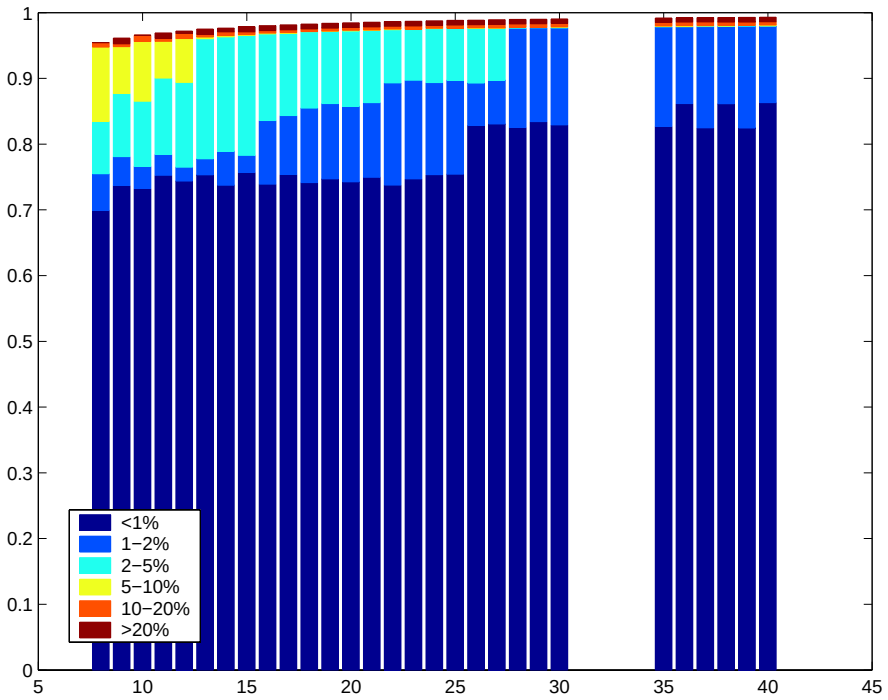
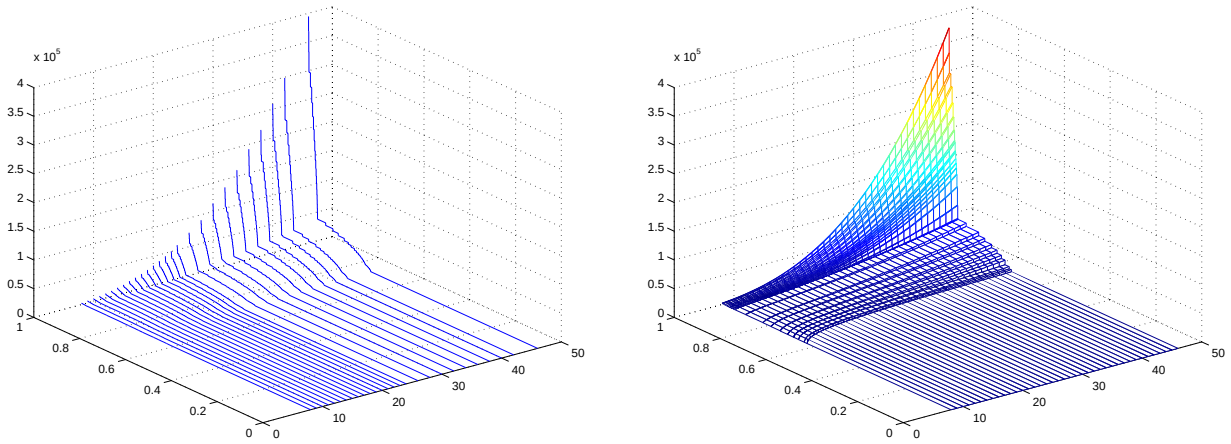


Figure 1: The error distribution of the average memory reuse distance model for Sweep3D. The x axis represents the problem size and the y axis represents the fraction of accesses. The model has less than 1% error for three quarters of the accesses and the tendency is to have a higher precision for larger problem sizes.

inaccurate for predicting the number of misses when we compared the results against a run on an SGI Origin 2000 using the hardware performance counters to measure the number of misses for the L1 and L2 caches. We realized that most memory instructions had a dual behavior: only a fraction of the accesses produced by that instruction were misses while our model was predicting either 100% hits or 100% misses for a given instruction. By using only the average reuse distance we lost a significant part of the information that we collected. One very large reuse distance averaged with several very short reuse distances resulted in a larger than cache size average, which yielded a prediction of all misses.

Our new approach for modeling the dynamic behavior of a memory operation in the machine code uses as input a compressed form of a complete histogram of reuse distances collected by our binary instrumentation. It first divides the accesses for an instruction into multiple bins and then computes a separate parameterized model for each bin. The algorithm for determining the number of bins and their size and structure uses a divide and conquer approach, recursively splitting a set of accesses in two until the two subsets have similar fitting curves. We apply the algorithm to the entire set of data and at each step we execute a similar division for the data sets corresponding to each problem size. We use a heuristic algorithm to determine how to partition the accesses. Its decisions influence the convergence speed, the accuracy and the stability of the final model. In our experiments, the partitioning heuristic that yielded the most stable and accurate results was one that selects partition boundaries so that the ratio between the pair of partitions resulting from a split are similar across all problem sizes. After partitioning, we perform a coalescing step that examines adjacent bins and aggregates them together if they have similar polynomials describing them. Each bin is modeled by two polynomials, one for the number of accesses that are part of that bin and one models how the average reuse distance of these accesses changes with problem size.

Figure 2(a) shows the reuse distance data collected by our tool for one of the most frequently executed memory accesses in Sweep3D and Figure 2(b) presents our parameterized model for that instruction. The `x` axis represents the problem size (from 8 to 46 in this case), the `y` axis represents the normalized number of accesses for each problem size and on the `z` axis is the reuse distance. Our model parameterized by problem size can be evaluated for an arbitrarily large problem size in a fraction of a second. The problem of determining the ratio of hits and misses for a given cache size `csize` is equivalent with determining the intersection of the surface with the plane defined by $z = csize$. Similarly, the problem of computing the expected behavior for one instruction at a given problem size `psize` is equivalent with determining the intersection of the surface and the



(a) Collected data

(b) Constructed model

Figure 2: The surface on the right is the parameterized model for one of the most frequently executed memory accesses in Sweep3D. On the left is the original data collected for that instruction.

plane defined by $x = psize$. We can also formulate the problem of determining the minimum cache size such that the hit-ratio is `h`. This problem is equivalent with the intersection of the built model and the plane defined by $y = h$. Any two of these three problems can be combined and the solution is the intersection of the surface with the corresponding two orthogonal planes.

The actual problem that we want to solve is predicting the ratio of misses for a given problem size and cache size.

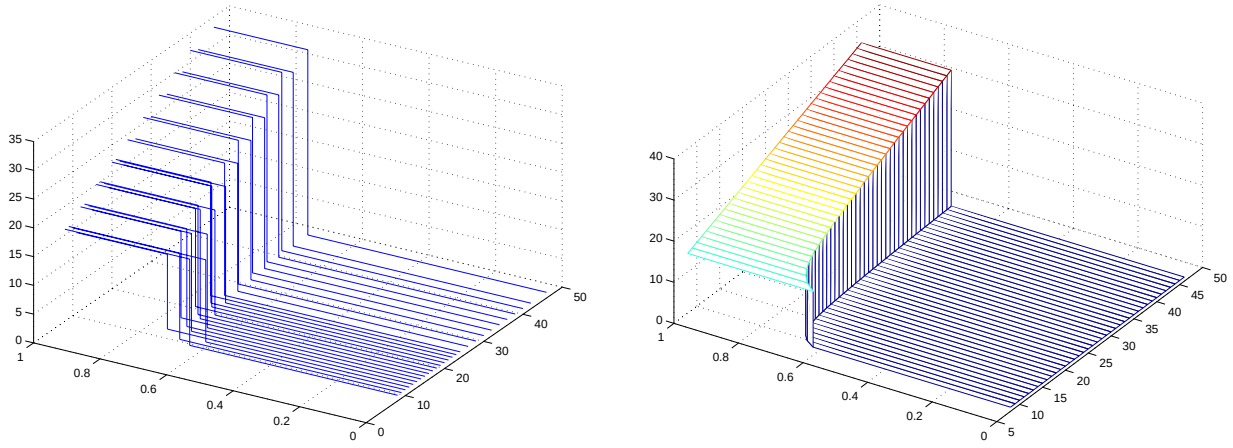
For the particular instruction in Figure 2 approximately 75% of the executed accesses see a small, constant memory reuse distance and therefore will be hits for any problem size. The other 25% of the accesses end up in several bins, each one having a distinctive monotonically increasing function for the reuse distance. The constant reuse distance seen by three quarters of the accesses is due to the spatial reuse in the cache. The cache line considered in the model is 32 bytes long and four elements (of type `double`) can be packed in one cache line. The first access to a cache line results in a large reuse distance and the next three stride-1 accesses to the same cache line will have a small reuse distance.

Many instructions have a more uniform distribution of their accesses' reuse distance. Figure 3 presents another frequently executed instruction from Sweep3D where about three quarters of the accesses have a small, constant reuse distance. In this case, the other 25% of the accesses are grouped in one single bin with a linear growth function for the reuse distance.

4 Partial Results

Current validation tests indicate that our tool produces accurate models for the expected reuse distance. How these models translate into accurate predictions of cache miss-ratios depends on the number of conflict misses the application inccurs. Conflict misses are inherent in today's caches with a reduced level of associativity and their number is strongly influenced by problem size and the layout of the data structures in memory. The models based solely on the reuse distance cannot predict the number of conflict misses and how they affect the application's performance. Furtunately, compilers and application writers can optimize programs to reduce to a minimum the number of conflict misses for a given architecture. Table 1 presents side by side the values predicted by our tool and the values measured with the hardware performance counters on an R12000 processor for a matrix-multiply binary. To assess the accuracy of our model in the absence of distortion caused by severe conflict misses, we transposed matrix B in order to increase the spatial reuse of the accesses to this data structure and to reduce the number of conflict misses. Although this transformation does not completely remove conflict misses, for the range of problem sizes presented in the table they are insignificant.

The results for this simple binary suggest that our method can be successfull in semi-automatically modeling the memory behavior of a black-box application.



(a) Collected data

(b) Constructed model

Figure 3: The collected data and the constructed model for another frequently executed memory access from Sweep3D.

Matrix size	Predicted Values			Hardware Counters Values		
	Num. accesses	L1 misses	% L1 misses	Num accesses	L1 misses	% L1 misses
500	2.513e+08	1.584e+07	6.306%	2.50e+08	1.59e+07	6.36%
853	1.245e+09	7.840e+07	6.297%	1.24e+09	7.92e+07	6.387%
1024	2.153e+09	1.364e+08	6.337%	2.15e+09	1.38e+08	6.418%
1271	4.115e+09	2.603e+08	6.327%	4.11e+09	2.66e+08	6.472%

Table 1: A comparison side by side of the predicted and measured values for a matrix-multiply application.