Dynamic Metrics for Compiler Developers

Sable Technical Report No. 2002-11

Bruno Dufour, Karel Driesen, Laurie Hendren and Clark Verbrugge

November 18, 2002

Contents

Abstract

In order to perform meaningful experiments in optimizing compilation and run-time system design, researchers usually rely on a suite of benchmark programs of interest to the optimization technique under consideration. Programs are described as *numeric*, *memory-intensive*, *concurrent*, or *object-oriented*, based on a qualitative appraisal, in some cases with little justification. We believe it is beneficial to quantify the behavior of programs with a concise and precisely defined set of metrics, in order to make these intuitive notions of program behavior more concrete and subject to experimental validation. We therefore define a set of unambiguous, dynamic, robust and architecture-independent metrics that can be used to categorize programs according to their dynamic behavior in five areas: size, data structure, memory use, concurrency, and polymorphism. A framework computing some of these metrics for Java programs is presented along with specific results.

1 Introduction

Understanding the dynamic behavior of programs is one important aspect in developing effective new strategies for optimizing compilers and runtime systems. Research papers in these areas often say that a particular technique is aimed at programs that are *numeric*, *loop-intensive*, *pointer-intensive*, *memory-intensive*, *object-oriented*, *concurrent*, and so on. However, there appears to be no well-established standard way of determining if a program fits into any of these categories. The goal of the work presented in this paper was to develop dynamic metrics that can be used to measure relevant runtime properties of programs, with the ultimate goal of establishing some standard metrics that could be used for quantitative analysis of benchmark programs in compiler research.

To be useful, dynamic metrics should provide a concise, yet informative, summary of different aspects of the *dynamic* behavior of programs. By *concise*, we mean that a small number of numeric values should be enough to summarize the behavior. For example, a complete profile of a program would not be a concise metric, whereas the fact that 90% of program execution is accounted for by 5% of the methods is a concise metric. By *informative*, we mean that the metric must measure some program characteristic of relevance to compiler developers, and the metric should differentiate between programs with different behaviors. For example, computing the ratio of number of lines of code to number of lines of comments does not capture anything about program behavior, and is not an informative metric, for our purposes.

In order to get a good overview of program characteristics of interest to compiler and runtime systems developers, we first studied the typical dynamic characteristics reported in papers presented over the last several years in the key conferences in the area. Although we did find many interesting experiments that suggest potential dynamic metrics, we also found that many summaries of benchmarks and results focused on static program measurements (#lines of code, # of methods, # loops transformed, # methods that are inlinable, # number of locations pointed to at dereference sites, and so on). Based on our own experiences, we suspect that this focus on static, rather than dynamic, metrics is at least partly because the static metrics are much easier to compute. Thus, one important goal of this paper is to provide both a methodology to compute the dynamic metrics and a database of dynamic metrics for commonly used benchmarks.

In our development of dynamic metrics we also discovered that different program behaviors need to be summarized in different ways. For example, if one is measuring the behavior of virtual method calls in an object-oriented language like Java, one summary might be the average number of target methods per virtual call site. However, compiler developers are usually most interested in the special cases where the virtual call is monomorphic (good for inlining) or perhaps corresponds to a very small number of target methods (good for specialization). Thus, a much more relevant metric might be what percentage of virtual calls correspond to 1, 2, or more than 2, targets. Thus, in this paper we define three basic ways of reporting dynamic metrics, *values*, *bins*, and *percentiles*, and we suggest which type is most appropriate for each situation.

Based on our overview of the literature, and our own experiences and requirements, we have developed many dynamic metrics, grouped under five categories: (1) size and structure of programs, (2) data structures, (3) polymorphism, (4) memory and (5) concurrency. In this paper we provide definitions for metrics in each category, and we provide specific examples of computing some of these metrics on well known benchmarks.

Computing the dynamic metrics turned out to be more difficult than we first anticipated. For this paper we have developed a framework for computing the metrics for Java programs. Our framework consists of two major pieces, a front-end JVMPI-based agent which produces relevant events, and a back-end which consumes the events and computes the metrics.

The major contributions of this paper include:

- We have identified the need to have dynamic metrics for compiler and run-time system developers and discussed why having such metrics would be of benefit.
- We provide an analysis of the different ways of presenting metrics and a discussion the general requirements for good dynamic metrics.
- We provide a detailed discussion of five groups of specific metrics that should be of interest to compiler writers along with specific examples of the metrics on benchmark programs.
- We present our framework for computing the metrics for Java programs.

The rest of this paper is organized as follows. Section 2 discusses the requirements of good dynamic metrics and Section 3 presents three different ways of presenting the metrics. In Section 4 we introduce five groups of dynamic metrics, with specific examples of metrics for each group. Section 5 summarizes the framework we used for collecting dynamic metrics and discusses limitations and further improvements to this approach. In Section 6 we provide an overview of related work and in Section 7 we give our conclusions and future work.

2 Requirements for Dynamic Metrics

Dynamic metrics need to exhibit a number of characteristics in order to render clear and comparable numbers for any kind of program. The following is a non-exhaustive list of desirable qualities.

- Unambiguous: one lesson learned from static metric literature is that ambiguous definitions lead to unusable metrics. For instance, the most widely used metric for program size is ’lines of code’ (LOC). LOC is sufficient to give a ball park measure of program size. However, without further specification it is virtually useless to compare two programs. Are comments and blank lines counted? What is the effect of indentation? How do you compare two programs from different languages? Within a given language, the LOC of a pretty-printed version of a program with comments and blank lines removed would give an unambiguous measurement that can be used to compare two programs.
- Dynamic: obviously a dynamic metric needs to be dynamic. In other words, *the metric should measure an aspect of a program that can only be obtained by actually running the program*. While this usually requires more work than a static measurement, the resulting numbers will be more meaningful since they will not change by adding unexecuted code to the program. Dead code should not influence the measurement. We will refer to instructions that are executed at least once as *instructions touched*, or live code.
- Robust: the other side of the coin of using dynamic measurements is the possibility that those numbers are heavily influenced by the program input. Where static numbers may be meaningless because non-executed parts contribute to the numbers, dynamic numbers may be meaningless because the particular execution that is measured may not reflect the common program behavior. Unfortunately, we simply cannot guarantee that a program’s input is representative. However, we can take care to define metrics that are robust with respect to the input. In other words, *a small change in a program’s input should cause a correspondingly small change in the resulting metric*. In particular, a robust metric should not be overly sensitive to the size of a program’s input. Total number of instructions executed is not a robust metric, since a bubblesort, for example, will execute four times as many instructions if the input size is increased by a factor of two. Number of *different* instructions executed is a robust metric since the size of the input will not drastically change the size of the part of the program that is executed.
- Machine-independent: since the metrics pertain to program behavior, *they should not change if the measurement takes place on a different platform* (including virtual machine implementation). For example, number of objects

allocated per second is a platform-dependent metric which disallows comparisons between measurements from different studies, because it is virtually impossible to guarantee that they all use identical platforms. On the other hand, number of objects allocated per 1000 executed bytecode instructions (kbc), is a platform-independent metric. In general, metrics defined around the byte code as a unit of measurement are machine-independent for Java programs.

3 Kinds of Dynamic Metrics

While there are many possible metrics one could gather, we have found that the most commonly described metrics, and the ones which seem most useful compiler optimization, tend to be belong to just a few basic categories. This includes the ubiquitous single value metrics such as average, more detailed continuous "expansions" of single value metrics, hot spot detection metrics, and metrics based on discrete categorization. It is of course possible to design and use a metric that does not fit into these categories; these initial metric kinds, however, enable us to at least begin to explore the various potential metrics by considering whether an appropriate metric exists in each of our categories.

3.1 Value Metrics

The first kind of metric we present is a standard, usually one value answer. Many data gatherers, for instance, will present a statistic like *average* or *maximum* as a rough indicator of some quantity; the idea being that a single value is sufficiently accurate. Typically this is intended to allow one to observe differences in behaviour before and after some optimization, smoothing out unimportant variations. For example, a value such as running time is perhaps best presented as an average over several executions. Value metrics appear in almost every compiler research article that presents dynamic measurements.

3.1.1 Continuous Value Metrics

Continuous value metrics are value metrics that have a continuous analogue, meant to be an expanded presentation of the value metric. E.g., a running average computed over time, or a count of file accesses presented over time.

Motivation for continuous value metrics arises from the inherent inaccuracy of a single value metric in many situations: a horizontal line in a graph can have the same overall average as a diagonal line, but clearly indicates very different behaviour. Additional descriptive values like standard deviation can be included in order to allow further refinement; unfortunately, secondary metrics are themselves often inadequate to really describe the difference in behaviour, requiring further tertiary metrics, and so on. Specific knowledge of other aspects of the metric space may also be required; correct use of standard deviation, for example, requires understanding the underlying distribution space of result values. Analysis situations in compiler optimization design may or may not result in simple normal distributions; certainly few if any compiler researchers verify or even argue that property.

In order to present a better, less error-prone metric for situations where a single value is potentially inaccurate, a straightforward solution is to present a graph of the value over a continuous domain (like time). Biased interpretations based on a single value are thus avoided, and an astute reader can judge the relative accuracy or appropriateness of the single value metric themselves. Continous value metrics can then be seen as an extension to value metrics, giving a more refined view of the genesis of a particular value.

3.2 Percentiles

Frequently in compiler optimization it is important to know whether the relative contributions of aspects of a program to a metric are evenly or unevenly distributed among the program elements. If a few elements dominate, then those can be considered "hot," and therefore worthy of further examination or optimization. Knowing, for example, that 2% of allocation sites are responsible for 90% of allocated bytes indicates that those top 2% of allocation sites are of particular interest. For comparison, a program where 50% of allocation sites contribute 90% of allocated bytes indicates a program that has a more even use of allocation, and so intensive optimization of a few areas will be less fruitful.

Similar metrics can be found in compiler optimization literature; e.g., the top *x*% of most frequently-executed methods [21].

3.3 Bins

Compiler optimization is often based on identifying specific categories of measurements, with the goal of applying different optimization strategies to different cases. A call-site optimization, for instance, may use one approach for monomorphic sites, a more complex system for polymorphic sites of degree 2, and may be unable to handle sites with a higher degree of polymorphism. In such a situation single value metrics do not measure the situation well, e.g., computing an average number of types per call site may not give a good impression of the optimization opportunities. An appropriate metric for this example would be to give a relative or absolute value for each of the categories of interest, 1, 2, or $\geq$3 target types. We refer to these kinds of metrics as “bins,” since the measurement task is to appropriately divide elements of the sample space into a few categories or bins.

There are many examples of bins in the literature; e.g., categorizing runtime safety checks according to type (null, array, type) [18], the % of loops requiring less than *x* registers [31].

4 Dynamic Metrics

In the previous two sections we have outlined three different kinds of dynamic metrics (value, percentile and bin), and some general requirements for good dyanmic metrics. In this section we present some concrete dynamic metrics that we feel are suitable for summarizing the dynamic behaviour of Java programs (although many of the metrics would also apply to other languages).

In developing our dynamic metrics we found that they fit naturally into five groups: (1) program size and structure, (2) measurements of data structures, (3) polymorphism, (4) dynamic memory use and (5) concurrency. In the following subsections we suggest specific metrics for each category.

4.1 Program Size and Structure

Dynamic metrics for program size and structure try to answer the question: how large is a program and how complex is its control structure?

4.1.1 Size

Before dynamic loading became commonplace, an approximation of this metric was commonly provided by the size of the executable file. With dynamic loading in place, the program has to be run to obtain a useful measurement of its size. We propose three metrics to characterize a program's run time size, which are progressively more dynamic.

size.load.value The number of byte code instructions loaded. Whenever a class is loaded, its size in byte code instructions is added to a running total. The standard libraries are excluded from this measurement, since they distort the numbers (java.*, javax.*, sun.*, com.sun.*). This is the closest equivalent to the static size of an executable. For Javac, 70K byte codes are loaded.

size.run.value The number of byte code instructions touched. This metric sums the total number of byte code instructions which are executed at least once in the entire duration of the program's execution. Run size is smaller than load size, since dead code is not counted. Javac touches 23K byte codes, or 31% of all loaded byte codes.

size.hot.percentile The number of byte code instructions responsible for 90% of execution. This metric is a percentile, obtained by counting the number of times each byte code instruction is executed, sorting the instructions by frequency, and reporting the number of (most frequent) byte codes which represent 90% of executed byte codes. Hot size is smaller than load size, and we expect it to be the most robust with respect to program input. 2240 byte codes are responsible for 90% of the execution of javac, 3% of all byte codes loaded, or 10% of all byte codes touched.

Of these three metrics we consider size.run.value the most important in characterizing program size. It is unambiguous, not influenced by dead code, robust and machine-independent. To further simplify, size.run.value can be mapped to a classification in five categories: XS,S,M,L,XL. A program printing "hello world", with 4 byte codes touched, is in the XS category, mtrt, with 7793 byte codes is in the M category, while javac with 23K byte codes is in the L category.

4.1.2 Structure

The following metrics characterize the complexity of program structure by measuring instructions that change control flow (if, switch, invokeVirtual). A program with a single large loop is considered simple, as opposed to a program with multiple loops and/or many control flow changes within a singe loop.

structure.controlDensity.value The total number of control byte codes touched divided by the total number of byte codes touched.

structure.changingControlDensity.value The total number of control byte codes that change direction at least once divided by the total number of byte codes touched. This measurement is smaller than the previous one, since many control byte codes never change direction.

structure.changingControlRate.value The number of changes in direction divided by the number of control instruction executions. This is the most dynamic measurement. It is the equivalent of the miss rate of the simplest dynamic hardware branch predictor which predicts that a branch will follow the same direction as the last time it was executed. The metric assumes that the branch history table, as this prediction scheme is known [20], is free from interference or capacity misses.

Of these metrics, structure.ControlDensity.value characterizes best the complexity of program structure. It is the reciprocal of average basic block size.

4.2 Data Structures

The data structures and types used in a program of of frequent interest. Optimization techniques change significantly for programs that rely heavily on particular classes of data structures; techniques useful for array-based programs, for instance are different from those that may be applied to programs building dynamic data structures.

4.2.1 Array Intensive

Many “scientific” benchmarks are deemed so because the dominant data structures are arrays. The looping and access patterns used for array operations are then expected to provide opportunities for optimization. Determining if a program is array intensive will then be a problem of determining if there are a significant, or dominant number of array accesses. Note that since Java multi-dimensional arrays are stored as arrays of arrays, the number of array access operations required for each multi-dimensional array element is magnified. To avoid skewing results, the type of element located at the array element should be inspected, and if it is an array type it should be discounted.

data.arrayDensity.value This is a metric describing the relative importance of array access operations. For uniformity, we express it as the average number of array access operations excluding accesses where the target value is not of array type, per kbc of executed code. A data.arrayDensity.value of greater than 100 indicates an array intensive program. A corresponding data.arrayDensity.continuous metric measures the same quantity incrementally, at various intervals in program execution.

As an example of array intensive, the data.arrayDensity.value for a simple (array-based) quicksort benchmark is 120, and for a very short FFT benchmark it is 100. The Jack benchmark of SPEC has a value of 30, and Javac has 40.

4.2.2 Floating-Point Intensive

Programs that do numerous floating point calculations also tend to be considered “scientific.” Different optimizations apply though; including the choice of appropriate math libraries optimized for speed or compatibility, opportunities for more aggressive floating point transformations and so on. Fortunately, floating-point operations are quite rarely used in most applications that do not actually focus on floating-point data, and so identifying floating-point intensive benchmarks is relatively straightforward from a simple ratio of instruction types.

data.floatDensity.value This single value describes the relative importance of floating-point operations. It is computed as the average number of floating point operations (including float and double types, and also loads and stores of float or double values) per kbc of executed code. A data.floatDensity.value greater than 100 indicates a floating-point intensive program. A corresponding data.floatDensity.continuous metric measures the same quantity at various program execution intervals.

For comparison, the data.floatDensity.value for Jack, Jess and Javac are all very small (< 10). Our integer-based quicksort also has a value < 10. Mtrt, however, uses floating point values extensively, and so has a value of 360; similarly FFT has a value of 310.

4.2.3 Pointer Intensive

Dynamic data structures are manipulated and traversed through pointers or object references. Programs that use dynamic data structures are thus expected to perform a greater number of object dereferences than a program which uses arrays as primary data structures; a basic metric can be developed from this observation. Of course a language such as Java which encourages object usage can easily skew this sort of measurement: an array-intensive program that stores array elements as objects (e.g., Complex objects) will result in as many object references as array references.

data.pointerDensity.value This gives a coarse indication of the importance of pointer references in a program. It is computed as the average number of loads of object references per kbc of executed code. The value 100 is used as a threshold for considering a program likely to be pointer intensive. A continuous version, data.pointerDensity.continuous is also easily defined.

The validity of this metric is apparent in the benchmarks we measured. FFT and quicksort have very low values (less than 20), whereas Javac has a value of 210, and Jess has 240. Interestingly, mtrt has the highest value of the SPEC benchmarks (290); we note that while mtrt is numeric, it uses octtrees internally for space representation (as opposed to arrays).

A potentially more accurate way of determining whether a program makes extensive use of dynamic, pointer-based data structures is to measure the graph diameter (length of longest path) of all data structures—a very small diameter indicates very “flat” data structures, where pointer traversals will be shorter; a larger diameter indicates the need for deeper traversals to locate data. Again this approximate measure can be easily defeated by particular programming strategies: an array containing objects that maintain references to their immediate grid neighbours will have a relatively high diameter, whether or not the references are necessary or even used.

data.pointerDiameter.value This metric describes the length of the longest path of each weakly-connected data structure. Data structure roots can be approximated from the same root sets used by garbage collectors.

Pointer polymorphism is typically measured as an average or maximum number of target addresses per pointer, and symmetrically number of pointers per target address (Cheng and Hwu argue that both are required for a more accurate measurement [8]). This can be computed as a value metric; a bin version can also be appropriate, and is defined following.

data.pointsToCount.value This along with the symmetric data.pointsFromCount.value measures the average number of distinct objects referenced by each object references and the average number of object references directed at each object respectively. Continuous version can of course also be defined.

data.pointsTo.bin A pointer analysis system is most interested in identifying pointers that can be directed at one address, possibly two, but further divisions are often unnecessary. A bin metric can provide a more appropriate view in this case. Each bin gives the percentage of object references that referenced 1 object, 2 objects, and $\geq$3 objects. The symmetric companion bin, data.pointsFrom.bin, has bins for the percentage of objects that had 1, 2 or $\geq$3 references.

4.3 Polymorphism

Polymorphism is a salient feature of object-oriented languages like Java. A polymorphic call in Java takes the form of an invokeVirtual or invokeInterface byte code. The target method of a polymorphic call depends on the run time type of the object receiving the call. In programs that do not employ inheritance, this target never changes and no call is truly polymorphic. The amount of polymorphism can therefore serve as a measurement of a program’s object-orientedness.

Some of the metrics have two variants. In the first variant, we take into consideration the number of receiver types, in the second we use the number of different target methods. There are more receiver types than targets, since two different object types may inherit the same method from a common super class. Some optimization techniques, such as class hierarchy analysis [12], optimize call sites with a restricted number of targets. Others, such as inline caching [13], optimize call sites with a restricted number of receiver types.

4.3.1 Polymorphism

polymorphism.callSites.value The total number of different call sites executed. This measurement includes monomorphic virtual method calls with a single receiver, but not static invokes.

polymorphism.receiverArity.bin This is a bin metric. We show the percentage of all call sites that have one, two and more than two different receiver types.

polymorphism.targetArity.bin As previous, but counting the number of different method targets instead of different receiver types. This metric is useful for compiler optimizations aimed at devirtualization: whenever a compiler can prove that a call site only has a single possible target, the call can be replaced by a static call or inlined [27]. This metric is more useful than a simple average of number of targets per call site, which is easily distorted by one heavily polymorphic call site.

polymorphism.receiverPolyDensity.value This metric shows the number of call sites that have two or more receiver types divided by the total number of call sites. It is simply the sum of the two last bins of polymorphism.receiverArity.bin.

polymorphism.targetPolyDensity.value As previous, but counting the number of different method targets instead of different receiver types. It represents the proportion of call sites which cannot be devirtualized by any static compiler, since they have at least two targets during execution.

polymorphism.receiverBiasMissRate.value This metric shows as a percentage how often a call is made with other than a call site's biased receiver type. This metric measures the effectiveness of the following optimization: for every call site, we count the frequency of each receiver type. Many call sites are biased towards one receiver type. The polymorphic call can be replaced by a test that compares the run time receiver with the bias type and jumps directly to the correct target method (which can be inlined) [2]. The metric shows the percentage of calls in which this test fails.

polymorphism.targetBiasMissRate.value As previous but measuring bias to method target instead of receiver type.

polymorphism.receiverCacheMissRate.value This metric shows as a percentage how often a call site switches between receiver types. In other words, how often is the receiver type different from the one that would be cached inline? Usually this metric is smaller than polymorphism.receiverBias.rate, but there are exceptions. For example, a single call site with two equally likely receiver types will have a polymorphism.receiverBias.rate of 50%, but if it alternates between those two receiver types, polymorphism.receiverCache.rate will be 100%. On the other hand, if it executes with one receiver type for half of the duration of the program and then switches to the other receiver type for the rest of the execution, polymorphism.receiverCache.rate will be close to 0%. This is the most dynamic measurement of polymorphism, and it represent the miss rate of a true inline cache.

polymorphism.targetCacheMissRate.value As previous, but the cache compares with method target instead of receiver type. This measurement represents the miss rate of an idealized branch target buffer [16]. It is always lower than the previous polymorphism.receiverCache.missRate, since targets can be equal for different receiver types.

Of these metrics, polymorphism.targetPolyDensity.value has the advantage that devirtualization studies often report this number. However, for a more dynamic measurement of run time polymorphic behavior, polymorphism.receiverCacheMissRate.value is preferred.

4.4 Memory Use

For considering the memory use of programs, we concentrate on the amount and properties of dynamically-allocated memory (memory use for the stack is related to the call graph metrics, and memory for globals is not usually a dynamically varying value).

4.4.1 Allocation Density

The first metric required is just a simple value metric to measure how much dynamic memory is allocated by the program, per 1000 bytecode instructions (kbc) executed, and there are two variations.

memory.byteAlloctionDensity.value Measures the number of *bytes* allocated per kbc executed. It is computed as the total number of bytes allocated by the program, divided by the (number of instructions executed/1000).

memory.objectAllocationDensity.value Similar to the previous metric, but reports the number of *objects* allocated per kbc executed.

Although these metrics give a simple summary of how memory-hungry the program is overall, they do not distinguish between a program that allocates smoothly over its entire execution and a program that allocates only in some phases of the execution. To show this kind of behaviour, there are obvious continuous analogs, where the number of bytes/objects allocated per kbc is computed per execution time interval, and not just once for the entire execution (memory.byteAllocationDensity.continuous and memory.objectAllocationDensity.continuous).

4.4.2 Object Size Distribution

memory.averageObjectSize.value The average size of objects allocated can be computed using the ratio of memory.byteAllocationDensity.value to memory.objectAllocationDensity.value. This metric is somewhat implementation dependent, as the size of the object header may be different in different JVM implementations.

Rather than just a simple average object size, one might be more interested in the distribution of the sizes the objects allocated. For example, programs that allocate many small objects may be more suitable for some optimizations such as object inlining, or special memory allocators which optimize for small objects.

memory.objectSizeDistribution.bin Object size distributions can be represented using this bin metric, where each bin contains the percentage of all objects allocated corresponding to the sizes associated with each bin. In order to factor out implementation-specific details of the object header size we use bin 0 to represent all objects which have no fields (i.e. all objects which are represented only by the header). In order to capture commonly allocated sizes in some detail, bins 1, 2, 3, and 4 correspond to objects using $h+1$ words, $h+2$ words, $h+3$ words and $h+4$ words respectively, where h represents the size of the object header. Then, increasingly coarser bins are used to capture all remaining sizes, where bin 4 corresponds to objects with size $h+5\ldots h+8$, bin 5 corresponds to objects with size $h+9\ldots h+16$, bin 6 corresponds to objects with size $h+17\ldots h+48$ and bin 7 corresponds to all objects with size greater than $h+48$. Note that the sum of all bins should be 100%.

4.4.3 Hot Allocation Sites

Another interesting program behavior is to determine if there is a small fraction of allocation sites which account for a large fraction of total bytes/objects allocated.

memory.allocatedBytesHotspot.percentile This metric reports the fraction of live allocation sites (where an allocate site is live if it corresponds to an invocation that executes at least once) corresponding to 90% of total bytes allocated. A low number indicates that relatively few allocation sites are responsible for most of the memory consumption.

memory.allocatedObjectsHotspot.percentile This metric is the same as the previous one, but reports the fraction of live allocation sites corresponding to 90% of the total objects allocated.

4.4.4 Object Liveness

Researchers interested in garbage collection are often interested in the liveness of dynamically-allocated objects. For example, the generational collection is potentially a good idea if a large proportion of objects have short lifetimes. For liveness metrics, time is often reported in terms of intervals of allocated bytes. For example, for an interval size of 10000 bytes, interval 1 ends after 10000 bytes have been allocated, interval 2 ends after 20000 bytes have been allocated and so on.

Object lifetimes can be estimated by forcing a garbage collection at the end of each interval, thus allowing one to find the amount of live memory after the collection, and to capture the death of objects that have become unreachable during that interval.

Based on the the amount of live memory at the end of each interval, we can compute the following two metrics.

memory.highWaterHeapSize.value This metric is computed as the maximum of all live memory amounts over all intervals.

memory.averageHeapSize.value This metric is computed as the average of the live memory over all intervals.

The highwater mark indicates how big the heap has to grow. The average tells us how big the heap is on average. If the average is much smaller than the highwater mark, then the program has some phase that is memory hungry, but other parts that are less memory hungry.

To compute interesting metrics about object lifetimes we define the *birth_time* of an object as the interval number in which it was allocated, the *death_time* as the interval number in which it was freed, and the *last_used_time* as the interval number in which the object was last touched. Each object then has a *total_lifetime* (*death_time* - *birth_time*), which is composed of two subintervals: *active_lifetime* (*lastused_time* - *birth_time*) and *dragged_lifetime* (*death_time* - *lastused_time*).

memory.objectLifetime.bin This metric reports the percentage of objects corresponding to each bin, where we have bins for lifetimes of 1, 2, 3, 4, 5…8, 9…16, 17…32 and greater than 32. If most objects have short lifetimes, then a generational collector may be useful. If many objects have very long lifetimes, then a collector that optimizes for long-lived objects may be preferred.

Another concept that is appearing in the garbage collection literature is the notion of the dragged objects [23, 25]. Dragged objects are those that are still reachable (live), but are never touched for the remainder of the execution. To define a metric that measures the amount of drag for a program we define the *useful real estate* for an object *o* as *active_lifetime(o)* $\times$ *sizeof(o)*, the *useless real estate* for object *o* as *dragged_lifetime(o)* $\times$ *sizeof(o)*, and the *total real estate* for *o* as *total_lifetime(o)* $\times$ *sizeof(o)*. We can then look at total useless real estate as a fraction of total real estate. If this is a significant fraction, then dragged objects may be a problem in this benchmark.

memory.uselessRealEstateFraction.value This metric measures the overall useless real estate. It is computed as the sum of the useless real estate over all objects divided by the sum of the total real estate for all objects.

The previous metric summarizes the uselessRealEstateFraction for all objects. If this fraction is high, then there exists a drag problem in the benchmark. We can also categorize objects by their individual individual uselessRealEstateFraction values using bins.

memory.uselessRealEstateDistribution.bin For this metric we use 10 bins, one bin for each of the following intervals, 0.00…0.10, 0.11…0.20, …, 0.91…1.00. Each bin counts the percentage of allocated objects with an individual uselessRealEstateFraction in that interval.

4.5 Concurrency and Synchronization

Optimizations focussing on multithreaded behaviour need to identify the appropriate opportunities. A basic requirement is to know whether a program does or can actually exhibit concurrent behaviour, or is it effectively single-threaded, executing one thread at a time. This affects the application of various optimization techniques, most obviously synchronization removal and lock design, but also the utility of other analyses that may be constrained by conservative assumptions in the presence of multithreaded execution (e.g., escape analysis).

Since the use of locks can have a large impact on performance in both single and multithreaded code, it is also useful to consider metrics that give more specific information on how locks are being used. A program, even a multithreaded one that does relatively little locking will obviously have a correspondingly reduced benefit from optimizations designed to reduce the cost of locking or number of locks acquired. Lock design and placement is also often predicated on knowing the amount of contention a lock experiences; this can also be exposed by appropriate metrics.

4.5.1 Concurrent

Identifying concurrent benchmarks involves determining whether more than one thread[1] can be executing at the same time. This is not a simple quality to determine; certainly the number of threads started by an application is an upper bound on the amount of execution that can overlap or be concurrent, but the mere existence of multiple threads does not imply they can or will execute concurrently.

Ideally, one would run the program using a very large number of processors, so as many threads as possible could be scheduled simultaneously. Appropriate metrics would then include a single value such as maximum number of concurrently active threads, or a bin describing relative timings for discrete levels of concurrency. Unfortunately, with massive multiprocessor machines still uncommon this approach is somewhat infeasible in practice.

concurrency.threadDensity.value A less accurate, but more easily computable metric is to consider the maximum number of threads simultaneously in the ACTIVE or RUNNABLE states. These are threads that are either running, or at least capable of being run (but which are not currently scheduled). In this way we do not require as many processors as runnable threads. Unfortunately, this will certainly perturb results: two short-lived threads started serially by one thread may never overlap execution on a 3-processor; on a uniprocessor, however, scheduling may result in all three threads being runnable at the same time. Of course there is considerable variation already permitted by Java's thread scheduling model—metrics in this area will necessarily have considerable variation

concurrency.threadDensity.bin The amount of code executed while another thread is executing is also of interest; it gives an indication of how "much" concurrent execution exists. Without sufficient processors this is a difficult quantity to measure; again we resort to coarser approximations based on active and runnable threads. This quantity is then calculated as % of kbc executed while there are specified levels of concurrency: 1, 2, $\geq$3 threads active or runnable.

4.5.2 Lock Intensive

An important criterion in optimizing lock usage is to know whether a program does a significant number of lock operations. This is quickly seen from the single value metric of an average number of lock operations per execution unit (bytecode, or time). Continuous versions of the same would enable one to see if the locking behaviour is concentrated in one section of the program, or is specific to particular program phases.

concurrency.lockDensity.value This single value metric gives average number of lock (`monitorenter`) operations per kbc. A value greater than 10 is expected to indicate a lock intensive program. A continuous version is also sensible.

concurrency.lockContendedDensity.value Adaptive locks can make use of knowing whether a lock will experience contention. This allows them to optimize behaviour for single-threaded access, but also to adapt to an optimized contended access behaviour if necessary [5]. Similarly, lock removal or relocation strategies will be better if they have information on which locks are (perhaps just likely) high, low or no-contention locks. A simple metric relevant to these efforts is to try and measure the importance of contention; this can be a value giving the average number of contended lock entry operations per kbc.

[1]Note that the JVM will start several threads for even the simplest of programs (eg one or more garbage collector threads, a finalizer thread, etc). When identifying concurrency by the number of existing threads it is necessary to discount these if every benchmark is not to be considered concurrent.

Unfortunately, the SPEC benchmarks (even mtrt) are not very lock intensive: they all perform a relatively insignificant number of lock operations, contended or not (other researchers have found similar results; e.g., in [1]). Providing analysis data for programs using a greater number of threads is part of our future work.

concurrency.lock.percentile Locks may be amenable to hot spot optimization—specific locks can be optimized for use by a certain number of threads, or code can be specialized to avoid locking. Whether high-use locks exist or not can be identified through a percentile metric, showing that a large percentage of lock operations are performed by a small percentage of locks; for our metric we define this as the percentage of locks responsible for 90% of lock operations. A similar metric for contended locks can also be defined: concurrency.lockContended.percentile

concurrency.lockContended.bin Knowing the relative number of contended and uncontended locks is also important—this allows us to determine if contention is concentrated in a few locks, or more evenly spread out. A bin metric can be used to classify locks and the relative number of lock operations; this metric will describe the relative percentage of lock requested while already held by 0, 1, or ≥ 2 threads.

5 System for Collecting Dynamic Metrics

An essential part of any metric development process being empirical validation of the data, we required a way of easily computing metrics for a variety of Java benchmark programs. This led to the development of a new tool, DynaMetricO (for *Dyna*mic *Metric*s for *O*ptimizations), which is designed to allow us to quickly implement and test dynamic metrics for Java applications. This section describes the design objectives, the implementation and the future work of our new framework in details.

5.1 Design Objectives

Computing dynamic metrics appears to be a deceptively simple task. In fact, the huge amount of data that has to be processed constitutes a problem by itself. Because we were aiming at developing an offline analysis tool, performance was not a critical issue, but ensuring the tool works in reasonable and practical time was nevertheless not a trivial endeavor. The main design objective that influenced the development of DynaMetricO was however flexibility—we needed a tool which would let us investigate dynamic metrics with a high level of freedom.

5.2 Design

DynaMetricO consists of two major parts: a Java Virtual Machine Profiler Interface (JVMPI) agent which generates event traces for the programs to be analyzed, and a Java back-end which processes the event traces and computes the values for the various metrics. This design allows the two ends of the framework to be used independently.

The JVMPI was selected as a source of trace data primarily because of the ease with which it is possible to obtain specific information about the run-time behaviour of a program, and also because it is compatible with a number of commercial virtual machine implementations. Unfortunately, there are a number of limitations that are imposed by using the JVMPI for collecting data, the most serious of which being the fact that it is currently not possible to obtain information about the state of the execution stack using this approach. Within JVMPI the only solution to this problem is to simulate the entire execution, which imposes far too much overhead for our purposes. JVMPI also has a complex specification, including a variety of restrictions on agent behaviour in certain circumstances: producing a correct agent is not necessarily trivial.

5.2.1 The JVMPI Agent

The main responsibility of the DynaMetricO JVMPI agent is to gather the actual profile data; a secondary goal is to ensure the trace size is manageable. We note that although the JVMPI interface is well-defined, it is not always well-implemented, and some amount of programming effort is required to ensure the data gathered is complete. For instance, the agent keeps track of all entity identifiers, and explicitly requests events from the JVMPI interface when it encounters an event containing an unknown ID (these can occur for events that happen prior to JVMPI initialization). To reduce the size of the trace file, the agent will collapse several instruction events together when they correspond to a contiguous sequence in the method's code; this is then further reduced by using a binary trace output format. Using these combined strategies complete traces for the SPEC benchmarks (using size 100) can easily be stored on a regular hard disk, as they only require several gigabytes of memory.

The agent also allows the user to specify which events and which of their fields are to be recorded in the trace file using a simple domain-specific language, which is then compiled to a compact binary representation and included in the header of the trace. This ensures that any consumer of the trace file will be able to tell exactly what information is contained in the trace, and determine if all of its requirements are fullfilled before proceeding to the analysis.

5.2.2 The Analysis Back-End

DynaMetricO's back-end is implemented in Java, and thus benefits from an object-oriented design. A `MetricAnalysis` base class provides all of the necessary components of a metric analysis, so that implementing a new metric only requires providing code for four methods which are associated with the initialization, computation, finalization and result outputting phases of the metric. A `Pack` class allows the user to organize the various analyses into a hierarchy, as `Pack`s can contain any number of other `Pack` objects and trace analyses, including but not restricted to metric analyses. This hierarchy also provides a strict ordering of the analyses, such that events are always propagated in the same, well-defined order. This is an important property since analyses can modify (and possibly replace) events in the trace before the next analyses in the sequence are applied to them. For example, one of the first analyses to be executed in the standard analysis library is the `BytecodeResolver` analysis, which determines the type of bytecode executed from its method identifier and offset. Various other standard analyses are also provided, such as one which keeps track of identifiers for the various entities present in the trace file, or a filter analysis that is able to filter out events based on the name of the class to which they are related (used for removing library events).

Since each trace analysis has a unique name within its package the hierarchy uniquely determines names for all of the analyses. These names can then be used from the command-line interface to manipulate the various options that are specific to each analysis or package instance. This allows the tool to be completely customizable via scripts.

A significant effort has been put into making the analysis back-end require no more than a single pass on the trace file, making it useable using fifo special files (or pipes) when the size of the trace is too large to be stored on disk.

6 Related Work

In developing our dynamic metrics we first studied a large body of literature for static metrics, many of which are covered in Fenton and Pfleeger's book [17]. Although some static metrics have a use for compiler developers (for example, a normalized measure of static code size measures the size of the input to an optimizer), we found that many static metrics were somewhat ill-defined, and that static metrics did not capture program behavior that may be of interest to compiler developers.

We then searched the recent compiler publications to get a feel for the types of dynamic metrics that would be useful, and also the sorts of dynamic measurements already in common use in the field. Thus, our work is both a formalization of many familiar concepts and a development of some new concepts and metrics.

In our literature overview we found that dominant data structures and data types are usually identified by hand. Although most researchers will give relevant qualitative descriptions of the benchmarks in their test suite (floating point, array-based etc), terminology is not standard and categorization is rarely justified quantitatively. Pointer-based programs, however, receive more direct attention. Average size of points-to sets are computed by several researchers in order to show efficacy of pointer-analysis algorithms [9, 11, 19]; the symmetric requirements of showing points-to and points-from are argued in [8].

Dynamic memory allocation is actually a very well studied area, and researchers in the garbage collection community have made many studies about the dynamic behavior of allocations. In this paper we have tried to distill out some of the most common measurements and report them as meaningful metrics (as opposed to profiles).

Size metrics in the literature are typically based on a static measurement of program size, often lines of code or size of the executable. We did not come across size metrics which are based on the number of instructions touched during execution. Program structure and complexity in terms of control instructions are often reported as a side-effect of hardware branch prediction studies [3, 7, 15, 29, 30]. Unfortunately, the missrates are usually incomparable, since the predictors use limited tables and therefore include capacity misses, distorting the metric.

Polymorphism metrics can be found in three areas: studies using static compiler analysis to de-virtualize object-oriented programs [2, 12, 27], virtual machine implementation papers reporting inline cache miss rates [13, 28], and indirect branch prediction studies reporting branch target buffer miss rates for object-oriented programs [15, 29]. The latter are also usually distorted by limited branch target buffer sizes.

Concurrency is rarely measured dynamically, though some researchers do measure number of threads started [4]. Other metrics we present for measuring concurrency are unique. Measurement of synchronizations is considerably more common, if generally consisting of absolute counts of synchronization operations [6, 18, 24]. More detailed breakdowns are sometimes given; e.g., in [5].

Since the SPECjvm98 benchmarks appear to drive a lot of the development and evaluation of new compiler techniques, several groups have made specific studies of these benchmarks. For example, Dieckmann and Hölzle have presented a detailed study of the allocation behavior of SPECjvm98 benchmarks [14]. In this paper they studied heap size, object lifetimes, and various ways of looking at the heap composition. The work by Shuf et. al also looked at characterizing the memory behavior of Java Workloads, concentrating, on the actual memory performance of a particular JVM implementation and evaluating the potential for various compiler optimizations like field reordering [26]. Li et. al. presented a complete system simulation to characterize the SPECjvm98 benchmarks in terms of low-level execution profiles such as how much time is spent in the kernel and the behavior of the TLB [22].

All of these studies are very interesting and provide more detailed and low-level information that our high-level dynamic metrics. Our intent in designing the high-level dynamic metrics was to provide a relatively few number of data points to help researchers find those programs with interesting dynamic behavior. Once found, more detailed studies are most certainly useful. We also hope that by providing standardized dynamic metrics for programs outside of the SPECjvm98 suite of programs, we can help to expand the number of programs used to evaluate compiler optimizations for Java.

Daly et. al. performed a bytecode level analysis of the Java Grande benchmarks which is somewhat in the same spirit as our work, in the sense that they were interested in platform independent analysis of benchmarks [10]. Their analysis concentrated mostly on finding different distributions of instructions. For example, how many method calls/bytecodes executed in the the application, and how many in the Java API library, and what is the frequency of executions of various Java bytecodes. Our focus is also on platform independent analysis, but we are concentrating on developing a wide-variety of metrics that can be used to find different high-level behaviors of programs.

7 Conclusions and Future Work

We have defined five families of dynamic metrics which characterize a program's runtime behavior in terms of size and control structure, data structures, polymorphism, memory use, and concurrency and synchronization. These metrics were designed to be unambigious, dynamic, robust and machine-independent. From this comprehensive set of metrics, a concise subset was distilled, which should enable a compiler researcher to tell at a glance if a particular program exhibits the kind of behavior he/she is interested in. We implemented a metric collection framework using JVMPI to compute most of the defined metrics.

We have focussed on dynamic metrics because they give a more accurate measurement of program behaviour. Static metrics are significantly easier to compute, and give a rough indication of the work required for a particular optimization. However, in order to estimate the actual *effect* of an optimization on performance dynamic metrics are naturally essential. In our view the extra effort required to gather dynamic information results in a much more relevant view of the program to compiler and runtime optimization developers.

The metrics we describe form a basic, and reasonably comprehensive set of measurements describing a wide variety of program behaviour. We have covered most of the major qualitative descriptions reported by other researchers in the literature; in this way we expect the benchmark categorizations we have defined to be immediately useful to other researchers.

We believe there is a tremendous need for rigorously defined program metrics. Many studies we examined reported various numbers that were either not well-defined or could easily be skewed by small changes in program input or measurement style. We intend our work here to be foundational, giving specific and unambiguous metrics to the compiler community. It is our hope that this will inspire other researchers to describe their benchmarks with more rigour, and will also provide validity for the qualitative judgements researchers employ when collecting a benchmark suite

We plan to build further upon this work in several ways:

- Characterize a large suite of benchmark programs using the metrics described. We are in the process of constructing a web site with iconic representations associated with a variety of benchmarks. These icons give a good impression of the relevant qualities of a benchmark, based on the actual quantitative metrics we have defined. Researchers are invited to visit the website `http://www.sable.mcgill.ca/metrics` to find evaluations of a variety of benchmarks (we have currently processed SPEC JVM98, and are actively working on more).
- Extend the framework, using an instrumented open source Java virtual machine to collect the information not accessible through JVMPI, such as object drag.
- Continue to refine and extend the set of metrics. Suggestions and criticisms from the compiler community are most welcome.

Acknowledgments

This work was supported, in part, by NSERC, FCAR and McGill FGSR. Special thanks to Marc-André Dufour for designing the icons for the metrics.

References

[1] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y.S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. Technical Report TR-99-76, Sun Microsystems, 1999.

[2] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In Pierre Cointe, editor, *ECOOP'96—Object-Oriented Programming, 10th European Conference*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166, Linz, Austria, July 1996. Springer.

[3] A.N.Eden and T.Mudge. The YAGS branch prediction scheme. In *Proceedings of the International Symposium on Microarchitecture*, pages 69–77, November 1998.

[4] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. Java without the coffee breaks: a nonintrusive multiprocessor garbage collector. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 92–103. ACM Press, 2001.

[5] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation*, pages 258–268. ACM Press, 1998.

[6] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 35–46. ACM Press, 1999.

[7] P. Chang, E. Hao, and Y. Patt. Target prediction for indirect jumps. In *Proceedings of the International Symposium on Computer Architecture*, pages 274–283, June 1997.

[8] Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 57–69. ACM Press, 2000.

[9] Trishul M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 191–202. ACM Press, 2001.

[10] Charles Daly, Jane Horgan, James Power, and John Waldron. Platform independent dynamic Java virtual machine analysis: the Java Grande Forum benchmark suite. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, pages 106–115. ACM Press, 2001.

[11] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 35–46. ACM Press, 2000.

[12] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Walter G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Åarhus, Denmark, Aug 1995. Springer.

[13] L. P. Deutsch. Efficient implementation of the Smalltalk-80 system. In *Conference record of the 11th ACM Symposium on Principles of Programming Languages (POPL)*, pages 297–302, 1984.

[14] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of ECOOP 1999, LNCS 1628*, pages 92–115, 1999.

[15] K. Driesen and U. Hölzle. Multi-stage cascaded prediction. In *EuroPar '99 Conference Proceedings, LNCS 1685*, pages 1312–1321, September 1999.

[16] Karel Driesen. *Efficient Polymorphic Calls*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston/Dordrecht/London, 2001.

[17] Norman E. Fenton and Shari Lawrence Pfleeger. *Software metrics : a rigorous and practical approach*. PWS Publishing Company, 1997.

[18] Sanjay Ghemawat, Keith H. Randall, and Daniel J. Scales. Field analysis: getting useful and low-cost interprocedural information. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 334–344. ACM Press, 2000.

[19] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the importance of points-to analysis and other memory disambiguation methods for c programs. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 47–58. ACM Press, 2001.

[20] John L. Hennessy and David A.Patterson. *Computer Architecture: A Quantitative Approach (Third Edition)*. Morgan Kaufmann, San Francisco, 2002.

[21] Chandra Krintz and Brad Calder. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 156–167. ACM Press, 2001.

[22] Tao Li, Lizy Kurian John, Vijaykrishnan Narayanan, Anand Sivasubramaniam, Jyotsna Sabarinathan, and Anupama Murthy. Using complete system simulation to characterize SPECjvm98 benchmarks. In *Proceedings of the 14th International Conference on Supercomputing*, pages 22–33. ACM Press, 2000.

[23] Niklas Röjemo and Colin Runciman. Lag, drag, void and use - heap profiling and space-efficient compilation revisited. In *Proceedings of the first ACM SIGPLAN International Conference on Functional Programming*, pages 34–41. ACM Press, 1996.

[24] Erik Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 208–218. ACM Press, 2000.

[25] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Estimating the impact of heap liveness information on space consumption in Java. In *Proceedings of the third International Symposium on Memory Management*, pages 64–75. ACM Press, 2002.

[26] Yefim Shuf, Mauricio J. Serrano, Manish Gupta, and Jaswinder Pal Singh. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 194–205. ACM Press, 2001.

[27] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 264–280. ACM Press, 2000.

[28] David Ungar. *The Design and evaluation of a high-performance Smalltalk System*. MIT Press, Cambridge, 1987.

[29] N. Vijaykrishnan and N.Ranganathan. Tuning branch predictors to support virtual method invocation in Java. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies ans Systems*, May 1999.

[30] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[31] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. Improved spill code generation for software pipelined loops. In *Proceedings of the ACM SIGPLAN ’00 conference on Programming language design and implementation*, pages 134–144. ACM Press, 2000.