Architecture-Independent Modeling of Intra-Node Data Movement

Eric Anger Sudhakar Yalamanchili
Georgia Institute of Technology
Atlanta, GA
{eanger,sudha}@gatech.edu

Scott Pakin Patrick McCormick
Los Alamos National Laboratory
Los Alamos, NM
{pakin,pat}@lanl.gov

ABSTRACT

A primary concern of future high performance systems is the way data movement is managed; the sheer scale of data to be processed directly affects the achievable performance these systems can attain. However, the increasingly complex but inherently symbiotic relationships between upcoming scientific applications and high-performance architectures necessitate increasingly informative and flexible tools to ensure performance goals are met.

In this work we develop a memory-hierarchy model that quantifies a given application's cache behavior. What makes this work unique is that we instrument code at compile time, gather architecture-independent data at run time using a generic memory-hierarchy model, and delay selecting a particular cache hierarchy (levels, sizes, and associativities) to a post-processing step, where cache performance can be derived rapidly without having to re-run a slow cache simulator. We show that this approach is capable of predicting cache misses to within 13% of what is predicted by a traditional, high-fidelity, but slow cache simulator.

1. INTRODUCTION

Reaching the performance goals of future exascale systems will require increasing complexity in both the way applications are written as well as how hardware systems are designed. Codesign of the software and hardware is heralded as the key instigator of performance improvement, where both sides provide feedback and guidance for tuning. However, this collaboration requires new tools to ensure that only useful information gets communicated without bogging down either side with unnecessary details. From the perspective of application writers, rapidly changing or preproduction hardware obfuscates understanding the way algorithmic decisions map to execution artifacts. Conversely, the necessity of codesign burdens architects with understanding the requirements these new applications put on the system. Both sides benefit from robust tools that provide insight into the characterization of the system but that abstract away implementation details. That is, application writers care about the *effect* their application has on execution, rather than mechanism, whereas architects require an understanding of the demands applications put on hardware, not the domain science that causes it.

Data movement throughout the system is a pervasive concern of both the hardware and the software, directly affecting the time and energy to solution. Forecasts project that data access at all levels of the memory hierarchy will be the limiting factor of performance [5]. While a major constraint on reaching exascale goals, data movement is a complex issue to address because of the interrelation of different design decisions. One needs to examine a large space of possible memory hierarchies to determine how different workloads will perform. Existing tools are not designed for rapid exploration of such large design spaces. Traditional cache modeling techniques have significant execution overhead, as well as conflating platform architecture artifacts with the application performance. Instead, *architecture-independent* analysis of data movement allows for greater insight into application characterization while allowing rapid design space exploration of the hardware. LLVM [26] is a modern compiler infrastructure with strong industry involvement. It supports an exhaustive range of programming languages and backends, through a unified Intermediate Representation (IR). LLVM IR provides a suitable platform for architecture- and language-independent application analysis.

This paper presents a methodology for architecture-neutral modeling of data movement in a way that functions as an abstraction over machine and application specifics, aiding codesign by separating concerns of hardware architects and computational scientists. We leverage Byfl [32], a performance-analysis tool implemented as a LLVM pass, as the vehicle for our data movement analysis. We describe the techniques and motivations behind the model construction as well as its performance. The rest of this paper is organized as follows. Section 2 walks through related work in the field and how it differs from this work. Section 3 describes the Byfl tool and the mechanisms by which it leverages the LLVM IR. Section 4 describes the motivation and design of our memory model. Our experimental results and presented and discussed in Section 5. Finally, we summarize our findings in Section 6.

2. RELATED WORK

Several different techniques have been proposed for adding instrumentation to applications. Binary instrumentation tools such as Pin [28] and DynInst [11] take compiled programs and inject instrumentation before runtime. Approaches higher up the stack include source-to-source transformation tools such as ROSE [35], which instrument applications statically during compilation. Work by Shao et al. [39] explores architecture-independent modeling of ISAs for application characterization, modifying a JIT compiler to output instrumentation data. Our work differs from these through its use of Byfl, a framework that sits between binary- and source-level instrumentation, injecting instrumentation code into the LLVM IR. This provides an abstraction of the execution environment while retaining run-time information.

Techniques for predicting intra-node data movement through the memory hierarchy have a long history. The most common approach is to use detailed simulation to quantify cache transactions [12, 19, 25]. Cache simulations are slow because they consider minute details of the cache structure, many of which may not even represent first-order performance effects. Analytical models represent the other end of the spectrum. These models are characterized by higher-level intuition about how application constructs affect performance, at the cost of lower accuracy [14, 21, 41]. Analytical models necessarily make assumptions about the execution environment to abstract away system variations. In the middle of the spectrum are *stack models* [3, 13, 29, 37, 40, 44], a level of abstraction informed by hardware and application parameters but not tied to them. These models use the concept of reuse distance [16] to describe access locality and reuse. However, most prior work takes the perspective of optimization, where memory hierarchies are modeled in an attempt to increase performance. Our work, in contrast, uses these techniques to focus on architecture-independent application characterization, where the general performance trends of an application can be represented across a wide range of architecture designs, allowing for increased understanding and feedback to both application writers and hardware designers. We propose building these models on top of a modern compiler infrastructure to provide a means for *architecture-independent* modeling of data movement, giving both sides of codesign greater flexibility and feedback.

3. BYFL

The tool we use for exploring applications' data-movement properties is called Byfl [32].[1] (The name stands for "*by*tes and *fl*ops," which is all that Byfl v0.1 measured.) Byfl's underlying philosophy is to provide *architecture-independent* application characterization, which it does in the form of "software performance counters". These are analogous to the hardware performance counters that one might access via a library such as PAPI [10] but (1) do not require support from the underlying hardware, (2) are not limited to scalar counters; they may produce histograms or other aggregate data, (3) produce the same values on all platforms, (4) can all be active simultaneously (i.e., no need for multiplexing due to limited counter numbers or conflicting counter types), (5) are measured precisely, not sampled, which is important for fine-grained measurements, and (6) are not self-perturbing. An example of self-perturbation is counting retired instructions with hardware performance counters. This requires retiring additional instructions, which perturbs the measurement. Byfl's goal is to present performance information in a manner that is meaningful to application developers.

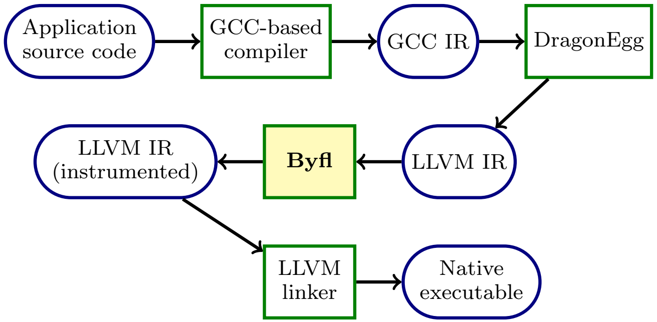


Figure 1: Byfl usage diagram

3.1 Architecture-independent performance analysis

To further motivate why it can be valuable to tally operations in a architecture-independent manner, consider comparing two versions of a code, for example to verify that an application and its associated mini-app [38] have similar resource utilization. In this scenario it would be hard to interpret measurements that indicate that code A performs more operations of some type (e.g., loads or stores) than code B on one architecture and indicate the reverse on a different architecture. Such measurements blur the *algorithm's* operation count with the idiosyncrasies of how that algorithm got mapped, register-scheduled, and peephole-optimized for a specific architecture.

While hardware performance counters serve an important role in optimizing code for a particular platform, our argument is that Byfl's software performance counters serve a complementary role in enabling algorithms to be analyzed and programs to be compared independently of instruction-set architecture or performance-counter semantics, which differ from platform to platform [18].

3.2 Implementation

Byfl instruments code at compile time but gathers data at run time. There are two main advantages to this approach:

1. Unlike binary-instrumentation tools such as Pin [28] and DynInst [11], Byfl has access to the compiler's view of the application, which retains more high-level aspects of the code (e.g., data types) than are available in the final machine code and omits architecture-specific operations that are not fundamental aspects of the algorithm (e.g., register spilling).

2. Unlike source-to-source transformation tools such as ROSE [35], Byfl has access to information that is not known at compile time (e.g., the number of iterations through a `while` loop or any value read from a file).

Figure 1 presents an overview of the Byfl instrumentation process. Byfl is implemented as an LLVM [26] compiler pass. It inputs LLVM's intermediate representation (IR) in LLVM bitcode format, injects instrumentation (counter increments and function calls), and outputs the instrumented LLVM bitcode. Although the Byfl concept could certainly be applied in the context of other IRs, such as GCC's GIMPLE [30], LLVM IR provides a convenient level of abstraction for the types of code transformations that Byfl performs, and the LLVM API makes it easy to apply these transformations.

[1] Byfl is freely available from `https://github.com/losalamos/Byfl`.

Listing 1: Shape function

```
 1 double apply_shape (int n, float *v, float *vs)
 2 {
 3   double accum = 0.0;
 4   int i;
 5
 6   for (i=0; i<n; i++) {
 7     float x = fabsf(v[i]);
 8     vs[i] = (1.0f − x)*(x <= 1.0f);
 9   }
10   for (i=0; i<n; i++)
11     accum += (double) vs[i];
12   return accum;
13 }
```

Listing 2: A Byfl-instrumented shape function

```
 1 vector.body: ; preds = %vector.body,
       %vector.body.preheader
 2   %index = phi i64 [ %index.next, %vector.body ],
         [ 0, %vector.body.preheader ]
 3   %8 = getelementptr inbounds float* %v, i64
         %index
 4   %9 = bitcast float* %8 to <4 x float>*
 5   %wide.load = load <4 x float>* %9, align 4,
         !tbaa !41
 6   %10 = tail call <4 x float> @llvm.fabs.v4f32(<4
         x float> %wide.load)
 7   %11 = getelementptr inbounds float* %vs,
         i64 %index
 8   %12 = fsub <4 x float> <float 1.000000e+00,
         float 1.000000e+00, float 1.000000e+00,
         float 1.000000e+00>, %10
 9   %13 = fcmp ole <4 x float> %10, <float
         1.000000e+00, float 1.000000e+00, float
         1.000000e+00, float 1.000000e+00>
10   %14 = select <4 x i1> %13, <4 x float> <float
         1.000000e+00, float 1.000000e+00, float
         1.000000e+00, float 1.000000e+00>, <4 x
         float> zeroinitializer
11   %15 = fmul <4 x float> %12, %14
12   %16 = bitcast float* %11 to <4 x float>*
13   store <4 x float> %15, <4 x float>* %16, align
         4, !tbaa !41
14   %index.next = add i64 %index, 4
15   %17 = icmp eq i64 %index.next, %n.vec
16   %gvar105 = load i64* @bf_flop_count, align 8
17   %new_gvar126 = add i64 %gvar105, 12
18   store i64 %new_gvar126, i64* @bf_flop_count,
       align 8
19   %idx_val153 = load i64* %idx_ptr, align 8
20   %new_val154 = add i64 %idx_val153, 1
21   store i64 %new_val154, i64* %idx_ptr, align 8
22   %idx_val157 = load i64* %garray, align 8
23   %new_val158 = add i64 %idx_val157, 1
24   store i64 %new_val158, i64* %garray, align 8
25   br i1 %17, label %middle.block.loopexit, label
         %vector.body, !llvm.loop !45
```

LLVM bitcode is essentially a high-level, canonical, RISC [33] assembly language that provides an infinite number of registers, employs strict typing, and represents all writes to registers in static single assignment (SSA) form [4, 36]. Byfl leverages these three features to count application rather than hardware features, to distinguish operations by data type if requested, and to easily identify valid locations in which to insert instrumentation code. An added benefit of instrumenting applications within a compiler is that the instrumentation code can be run through the same code optimizer as the application, which lowers the run-time cost of instrumentation.

As a simple example of Byfl instrumentation, consider tallying the number of floating-point operations performed by the C code shown in Listing 1, a code snippet representing the essence of the shape function used in the PlasmaApp particle-in-cell code [34]. The code maps a function across all elements of a vector (lines 6–9) using single precision for speed, then reduces those values (lines 10–11) in double precision to minimize error.

The LLVM bitcode in Listing 2 shows how Byfl tallies floating-point operations. The bitcode represents the first loop of Listing 1, lines 6–9. Floating-point operations (the **fsub**, **fcmp**, and **fmul** in lines 8, 9, and 11) are colored in fuchsia. Note that (a) the C `fabsf()` call was converted to a call to the LLVM `llvm.fabs.v4f32` intrinsic (line 6), which is considered a function call, not a floating-point operation—an arbitrary but at least a consistent decision—and (b) the compiler autovectorized the code to use four-element **float** vectors.

Byfl's additions are highlighted in lines 16–18 and represent loading the global `bf_flop_count` variable into a register, incrementing the register by 12 (3 floating-point instructions × a vector length of 4), and storing the new value back to `bf_flop_count`. We note that this code represents the instrumentation after optimization. The Byfl pass originally inserted {load, add 4, store} statements immediately after each floating-point operation; LLVM's optimizer automatically coalesced these into a single {load, add 12, store} for improved performance—another advantage of using LLVM IR for this work.

For convenience, Byfl comes with a set of wrapper scripts that simplify instrumentation. `bf-gcc`, `bf-g++`, `bf-gfortran`, and `bf-gccgo` wrap, respectively, the GNU C, C++, Fortran, and Go compilers. `bf-mpicc`, `bf-mpicxx`, `bf-mpif90`, and `bf-mpif77` wrap the similarly named Open MPI [22] and MPICH [23] wrapper scripts to use the preceding Byfl compiler scripts instead of the default C, C++, and Fortran compilers. These wrapper scripts take care of generating LLVM bitcode and running the Byfl compiler pass on it, all behind the scenes. A user need only modify his build process to replace the compiler and linker with the appropriate Byfl wrapper script.

Once an executable is produced, the user runs it as normal. During execution, each instrumented process outputs its measurements to a file. For the user's convenience, Byfl includes some post-processing scripts to convert its output into a form usable by KCachegrind's [43] or HPCToolkit's [2] graphical user interface.

3.3 Sample output

Table 1 lists the information that Byfl can currently out-

Table 1: Counters and other measurements currently implemented by Byfl

Counter	Program	Function	BB
Bytes loaded	✔	✔	✔
Bytes stored	✔	✔	✔
Flops	✔	✔	✔
Integer ops	✔	✔	✔
Loads	✔	✔	✔
Stores	✔	✔	✔
Unique bytes loaded/stored	✔	✔	
Conditional/indirect branches	✔	✔	
Function-invocation counts (even to non-Byfl-instrumented callees)	✔	✔	
Unconditional branches	✔		
Median and MAD reuse distance	✔		
Vector length/type/tally	✔		
Loads/stores by datatype and size	✔		
Instruction-mix histogram	✔		
Memory-usage histogram	✔		

put. At compile time, a user specifies which subset of this information should be maintained. (Naturally, the instrumented application runs faster if it does not need to keep track of as much information.) As Table 1 indicates, Byfl can work at the program, function (or, alternatively, function call stack), and/or basic-block granularity. Instrumentation can be enabled or disabled at the module or function level. Developers can also insert Byfl "calipers" into their code to explicitly enable and disable instrumentation at arbitrary points and to bin measurements by program-defined tags.

Not shown in Table 1 but discussed and explained in a prior publication [32] are two counters that are unique to Byfl: op bits and flop bits. These provide an alternative view of an application's balance between computation and data movement.

New features added to Byfl since our prior publication (and shown in Table 1) include the function-invocation counts; tallies of vector operations by length, type, and tally; tallies of loads and stores by datatype and size; instruction-mix histogram; reuse-distance data; and memory-usage histogram. From a data perspective, the noteworthy features in Table 1 are the reuse-distance data and the memory-usage histogram. Both are ways of providing architecture-independent views of data locality. Reuse distance [16], the median distance between repeated accesses to the same memory address, is slow to calculate but provides a meaningful lower bound on the amount of memory (or cache) needed to fit an application's working set. The work we describe in this paper is essentially an enhancement of the reuse-distance concept. The memory-usage histogram shows the minimum memory (or cache) size needed to contain 5%, 10%, 15%, … 100% of an application's dynamic memory accesses. It is much faster to compute—it is a simple per-address tally that is sorted on output—but lacks temporal information. That is, it cannot distinguish, for example, between access patterns $\{A, B, C, A, B, C, A, B, C\}$ and $\{A, A, A, B, B, B, C, C, C\}$. Again, this is rectified by the work we present in the following section.

Table 2: Example of Byfl output

Value	Metric
4,376,254	flops
97,813,342	integer ops
28,379,185	memory ops (23,671,118 loads + 4,708,067 stores)
22,099,736	branch ops (4,397,938 unconditional and direct + 12,792,872 conditional or indirect + 4,908,926 other)
189,271,197	bytes (166,772,045 loaded + 22,499,152 stored)
4,639,120	bytes stored by 150,716 calls to `memset()`
586,492	bytes loaded and stored by 40,330 calls to `memcpy()` or `memmove()`
358,580	vector operations (FP & int)
2.0000	elements per vector
63.9999	bits per element

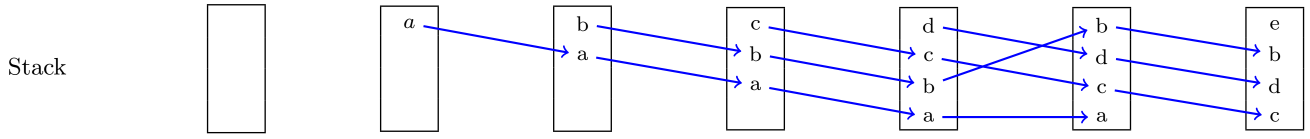
To make Byfl's behavior more concrete, Table 2 lists some information produced by a prototype of the HPCG benchmark [17] written in Legion [8]. When the implementation is complete, a Byfl comparison of the Legion version of HPCG to the original C++ version would be instructive to quantify the memory-usage differences introduced by Legion's data-centric parallel-programming model. It is clear from Table 2, however, that this code performs vastly fewer floating-point operations than branch, memory, or integer instructions; it requires substantial data traffic per floating-point operation (43.25 bytes/flop), and it currently vectorizes poorly, with only 0.35% of its flops and integer ops being vectorized.

4. THE STACK MODEL

Techniques for modeling caches have existed for over forty years in the form of software-managed page caches [29]. Conservative solutions use cycle-level simulations of cache structures, representing all mechanistic processes involved in increasingly complex microarchitectures [12]. Simulations are typically slow because their high degree of fidelity requires substantial processing. Rather than detailed representations of the architecture, this work employs a *stack model* to approximate the performance of the memory hierarchy.

During instrumentation, Byfl injects calls to the stack model, passing the address and size of each load and store operation. The stack retains, for each moment in time, how recently each unique address has been used. Caches typically implement a Least Recently Used (LRU) replacement policy, a common approximation of (non-implementable) optimal replacement, in which the line that will not be accessed for the greatest length of time is replaced [9]. The assumption is that a piece of data has a higher likelihood of reuse the more recently it was accessed. Our stack model transparently represents this behavior: the less recently a line has been used, the farther down the stack it resides. Consider the trace processing shown in Figure 2; newly received addresses are pushed onto the stack one at a time. The distance from the top of the stack to the currently requested line is known as its *stack distance* or *reuse distance* Δ_t [16], where t is the index in the address trace $X = x_1, x_2, \ldots, x_M$. If the line has not yet been referenced (when it does not reside in the stack), the reuse distance is defined as ∞. The reuse distance gives us a measure of access locality.

Time t	0	1	2	3	4	5	6
Address X_t	—	a	b	c	d	b	e



Reuse Distance Δ_t	—	∞	∞	∞	∞	3	∞

Slot C	Distance Count F						
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	1	1
4	0	0	0	0	0	0	0
∞	0	1	2	3	4	4	5

Figure 2: Sequence of requests to the stack model with four slots ($C = 4$).

Listing 3: Loop exhibiting temporal locality

```
for(int i = 0; i < 10; ++i){
    B[i] = A[i] + 8;
    ...
    C[i] = 3 * A[i];
}
```

The relationship between reuse distance and LRU cache size is implicit; the minimum size a cache must be to capture the reuse of a line is exactly Δ_t. Modeling a cache with C lines can be done with a stack model containing C slots. Every time a new request comes in, the stack is walked down from the top until either the line is found or the bottom is reached. The stack distance is counted as F and if the line was found it is removed from its old location. The line is pushed to the top of the stack, popping the last element if necessary to retain the total number of lines C. The sum of finite distance counts in F represents the number of successful accesses to the cache. The number of infinite reuse distances represents the number of cold misses.

4.1 Locality and Line Size

Caches take advantage of two types of locality in data accesses: *temporal* and *spatial*. Temporal locality is the result of accessing recently requested addresses again in the near future. Take for example the loop shown in Listing 3. In this loop, the updates to $B[i]$ read in the value of $A[i]$. Later on, the value of $A[i]$ is required to update $C[i]$. If the cache were to evict $A[i]$ after its first use, the second request will incur the penalty of an access to memory. Instead, caches have replacement policies like LRU that try to optimize for this type of locality under the expectation that there is a higher likelihood of a recently used line to be needed again in the future.

Spatial locality comes from the observation that addresses nearby recent requests have a high likelihood of use. Consider the loop in Listing 4; all three arrays are walked through in order, touching one element in each at a time. One may assume that, after accessing an array's value in one iteration, the subsequent value in the array will be accessed next.

Listing 4: Loop exhibiting spatial locality.

```
for(int i = 0; i < size; ++i){
    C[i] = A[i] * B[i];
}
```

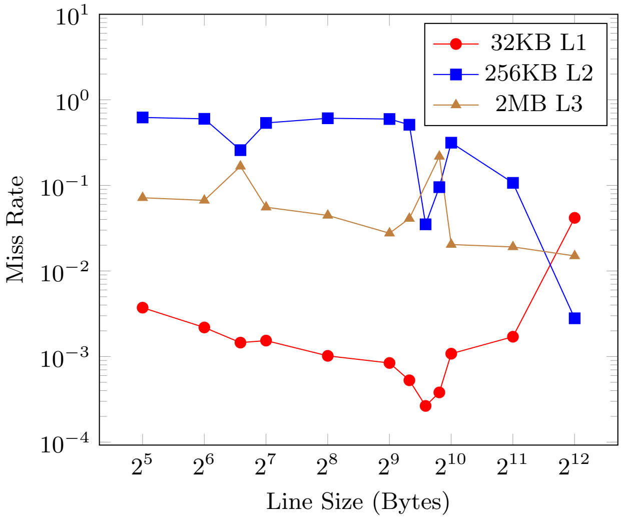


Figure 3: The effect of line size on miss rate.

A naïve cache would pull in a single value at a time, and have to make a request to memory for each access. A more intelligent design would pull in multiple values at a time so that subsequent requests would hit in the cache. This smallest addressable unit in the memory hierarchy is called a *line* or *block*. In the ideal case, every vector element would be pulled into the cache at once, so that only a single cache miss would occur. However, such long lines may bring in unused data, taking up part of the limited cache space; the appropriate size of a cache line is a design trade-off made by hardware architects. Figure 3 shows how miss rates can change as the line size changes. We ran the CoMD [20] application through the cache model described below for each line size, calculating the miss rates for each level of a three-level hierarchy.

4.2 Infinite Stacks

The stack model of size C can estimate the number of successful accesses to a cache with C lines. For a single reference stream, as a corollary of the LRU replacement nature of the stack model, the C most recently referenced lines are by definition contained in the $C+1$ most recently referenced lines. This can be seen by construction: with $C = 1$, the stack maintains the last line to be accessed. Loading a different line would then be pushed to the top of the stack, evicting the original line. If this stack were to have size $C = 2$, the previous head of the stack would be shifted down instead of evicted, reflecting the fact that it is the second most recently accessed unique line. Due to the relationship between reuse distance and critical cache size, all accesses to a cache size $C >= \Delta_t$ will capture the reuse. Rather than pinning the stack at a fixed size C and dropping all lines that have reuse distances $\Delta_t > C$, we grow the stack by one whenever a new line is accessed. This allows, for all accesses, the stack to maintain the *full* history of reuse. The distance count F is a histogram of reuse distances across theoretical stack sizes $0 < C <= N$ where N is the total number of unique lines touched by the application. Keeping a histogram provides architecture-independence: the total cache size may be selected *after* execution. All hits to a cache size C will be guaranteed to hit in all caches larger than C, so the total hit count, H_C, to a cache size C can be calculated as

$$H_C = \sum_{i=0}^{C} F_i \tag{1}$$

Multilevel caches, another major architectural design choice, are used to further increase the benefits of caching [6]. In these schemes more than a single cache is used; when a request misses in the cache closest to the core, the access is propagated down to the next largest cache, and so on as necessary until it finds the line in question, which may only reside in memory. Each level of cache in effect filters out some number of requests, propagating only on a miss. Modeling this effect requires the *inclusion property* of multilevel caches, where the entire contents of a smaller cache are contained in a larger cache [6]. The infinite stack model becomes an extension of this property: when a line misses in cache L_i, the request continues up to cache L_{i+1} if it exists, or the main memory. Any request that is not in cache L_i will not by definition reside in caches $\{L_j : j < i\}$ due to the inclusion property. The number of hits H_i to a single level i of the cache hierarchy with A total memory accesses is

$$H_i = \begin{cases} A - \sum_{k=0}^{C_i} F_{C_i} & i = 1 \\ \sum_{k=0}^{C_i} F_{C_i} - H_{i-1} & i > 1 \end{cases} \tag{2}$$

4.3 Associativity Counters

So far the stack model assumes that all recently accessed lines are guaranteed to reside in the cache. While true in a perfect cache, hardware constraints limit the practicality of this approach. Searching each slot within the cache for a potential hit is a time- and resource-intensive process. To combat this, modern caches utilize *associativity* to partition the total cache space into smaller, more manageable pieces. There are two main trade-offs at play: the desire to maximize effective capacity and the need to ensure timely searches for a line in the set. On one end of the spectrum is a *fully associative* cache, in which every line can reside anywhere in the cache. This maximizes potential cache space utilization, as a line has the best chance at avoiding conflict. However, the search function now has to check every line to see if it matches—a potentially expensive function as the size of the cache grows.

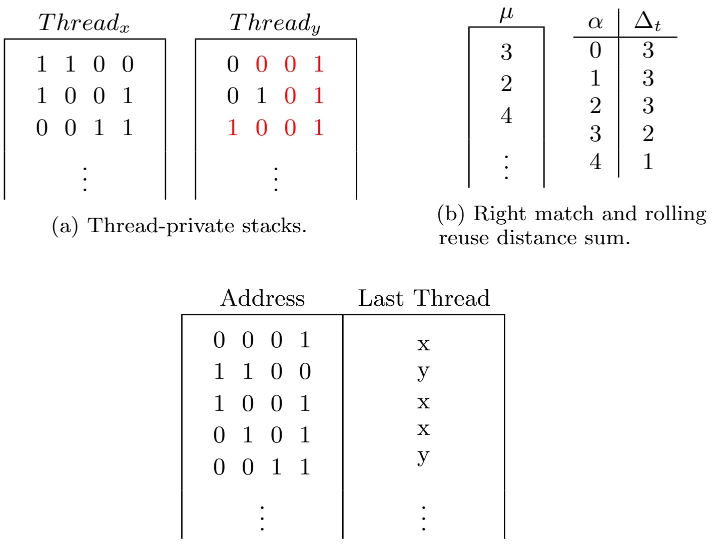
On the other end of the spectrum is the *direct-mapped* cache, wherein the mapping function relates a line's address to a single location within the cache. The overhead of this lookup is merely the cost of the hashing function, which is typically implemented as a subset of the address bits, called the *index*. However, it severely limits the effective space within the cache; if two lines happen to share the same mapping, they will compete over the same location instead of spreading out to an unoccupied slot. The middle ground is a *set-associative* cache, where lines may reside only in a single *set* containing a limited number of possible locations called *ways*. Here the cost of finding a line comprises the hash function and a search limited to the total number of ways. A *conflict miss* occurs when a cache access misses a line that was evicted in a set-associative cache that would not have occurred in a fully associative cache. Figure 5 illustrates the cost of associativity, viewed as a penalty on top of a fully-associative cache, for each level of the memory hierarchy on a run of the CoMD application. The application was run through the cache model described below.

We can naïvely model associativity by assigning each set its own stack. When a new request is made the appropriate stack is selected using the hash function, the reuse distance is calculated for that stack, and the line is pushed to the head position. However this technique depends on the number of sets being established a priori, tying the model to a specific architectural choice. Instead, we model all levels of associativity by leveraging the LRU composition of the stack; a single, unified stack aggregates all the information set-specific stacks contain, including the occurence of set conflicts, since the ordering of references to each set's stack is maintained.

This mechanism [24, 29, 44] captures arbitrary associativity with a set of reuse-distance counters per set size. The key is to reinterpret reuse distance. So far it has been a calculation of the number of unique addresses since a line was last accessed. We can imagine this being the case for a fully associative cache, where a new line is guaranteed a position if there is one available, otherwise the least recently used line is evicted. Associativity limits the number of lines that can separate a line from its last access. Two addresses in different sets never interact with each other, and therefore do not contribute to each other's reuse distance.

Modeling associativities requires a single counter per bit in the index. Figure 4 illustrates how, when a new line is requested, we walk down the stack until a matching line is found. At the same time we tabulate the *right match* of the two indices, defined as the number of matching index bits starting from the right (the least significant bit), as shown in red. A right match value of μ implies that a set must contain a minimum of 2^μ slots for the two lines to conflict. The reuse distance for 2^α sets, shown in Figure 4(b) is given by

$$\Delta_t^\alpha = \sum_{i=\alpha}^{k} \mu_i \tag{3}$$



$Thread_x$			
1	1	0	0
1	0	0	1
0	0	1	1
	⋮		

$Thread_y$			
0	0	0	1
0	1	0	1
1	0	0	1
	⋮		

(a) Thread-private stacks.

μ
3
2
4
⋮

α	Δ_t
0	3
1	3
2	3
3	2
4	1

(b) Right match and rolling reuse distance sum.

Address	Last Thread
0 0 0 1	x
1 1 0 0	y
1 0 0 1	x
0 1 0 1	x
0 0 1 1	y
⋮	⋮

(c) Shared stack.

Figure 4: Example thread-private (a) and global (c) infinite stacks for two-thread system upon $Thread_y$ receiving address 1001. Figure 4(b) shows right match and associativity counts for this access to $Thread_y$. Since $Thread_x$ was the last to access that address, this access would result in a remote cache access.

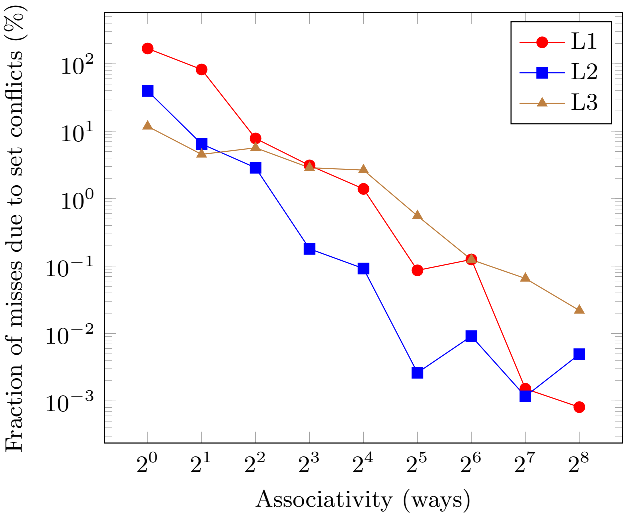


Figure 5: Overhead incurred by conflict misses from set associativity.

Here k is the number of bits in the index. Intuitively, two lines mapping to the same set with α set bits will also map to the same set with $\alpha - 1$ set bits. This rolling sum represents the number of unique lines that reside in this set, defining a minimum set size to ensure this reuse is captured by the cache. Consider the extreme case when there is only one set, $\alpha = 0$ (the fully associative case). All lines reside in the same set, degenerating to the total number lines from the top of the stack and the same result as the original stack model. Whenever a stack distance of Δ is measured, the minimum size the cache must be to capture that reuse is $C = 2^\alpha * \Delta$. If the line is not found, the reuse distance for all associativities is defined as ∞.

4.4 Multi-core Caches

The stack model described so far works only for uniprocessor computer architectures. Multiple cores residing in the same memory hierarchy incur changes to data movement patterns to ensure correctness and improve performance. This section describes additions to the stack model to accurately represent the effects of multi-core architectures.

A key concern is the way shared pieces of data are handled by the cache system. Imagine a two-level memory hierarchy, where the cores have their own L1 but share the L2 cache. Any data that passes into the core must come through the private cache that no other core can modify; all lines must be passed on a global bus connecting the caches. Core i may request a load of line r from the L2. The line gets transfered through the cache and resides in the lowest level cache private to i, $C_{1,i}$. Core j may also request line r, at which point the line will be copied to $C_{1,j}$. This allows both caches to keep a read-only copy of line r in their own private caches. As soon as core i modifies the line, it must broadcast an *invalidation* of r on the bus, saying that r is in a *dirty* state and all other copies of that line are out of date. Cache $C_{1,j}$ receives the message and marks line r as invalid. Should core j attempt to load that line again, it would find the line invalidated in $C_{1,j}$ and would need to push the request onto the bus. At this point cache $C_{1,i}$ would provide a copy of r, and both $C_{1,i}$ and $C_{1,j}$ would again have read-only copies.

Invalidations change access patterns in two ways. First, access to a line may result in a miss even if the reuse distance is smaller than the size of the cache if another core sends out an invalidation before that line is reused. Second, requests for a line may be serviced not from a higher level of the hierarchy but from a sibling core's cache at the same level if that line has been modified. In both of these cases, the total number of hits will differ from the number projected by the stack model as described so far.

To represent coherence traffic, we augment the stack model as shown in Figure 4. Each thread keeps its own stack to capture all requests to private cache structures. In addition, we maintain a single, global stack that captures requests from all threads, interleaved in the order they are issued. All lines in the stack remember which thread requested it. Processing a request takes the following form:

1. Walk down the private stack, incrementing right-match values as in the single-threaded case, until the line or the bottom of the stack is found.

2. Record the private reuse distance $\Delta_t^{private}$ for all associativities in $F^{private}$.

3. Move the line to the top of the private stack.

4. Walk down the shared stack, tabulating right match values.

5. If the line is found and the tagged thread ID differs from the current thread ID, record the reuse distance in F^{shared}. Add $\Delta_t^{private}$ to a separate count $F^{invalidated}$ to keep track of which hits would not occur in the case of invalidations.

6. Move the line to the top of the shared stack, updating the last thread access ID to this thread.

The total number of accesses that hit in n remote thread L1 caches is calculated as

$$H_{1,remote} = \sum_{j=0}^{n*C_1} F_j^{shared} \tag{4}$$

The total hits H_1 across all threads must then include these accesses:

$$H_1 = H_{1,remote} + \sum_{k=0}^{n}\sum_{j=0}^{C_1}(F_{k,j}^{private} - F_{k,j}^{invalidated}) \tag{5}$$

Any hits in remote L1 caches are then subtracted from the total hits to the L2 cache.

5. EXPERIMENTS

We illustrate the capabilities of our model using two exascale mini-applications. Specifically, we show that our model is capable of

- approximating the performance of traditional cache simulators across cache sizes,

- capturing the effect of set associativity, and

- predicting multithreaded execution artifacts.

A single compute node was used for all the experiments, as described in Table 3. While inter-node data movement is of interest, it is beyond the scope of this work. We assign only a single thread to each processor core, and model only the data cache.

This work compares the output of the proposed stack model with hardware performance counters taken from runs on real hardware. These are model-specific registers implemented in the processor to measure events that occur during execution. The counters are measured with the Linux `perf` tool [1], as shown in Table 4.

CoMD [20] is a molecular dynamics proxy application in which atoms exert forces upon one another, used to study the dynamical properties of liquids and solids used in the fields of materials science, chemistry, and biology. It is a highly-efficient code written in C, using MPI [31] for inter-node communication and OpenMP [15] for intra-node parallelism.

Table 3: Experimental machine setup

Parameter	Value
CPU model	Intel i7-4770k
Cores	4
Clock speed	3.5 GHz
Line size	64 bytes
L1 cache	4x 32 KB 8-way
L2 cache	4x 256 KB 8-way
L3 cache	Shared 8 MB 16-way
Memory	16 GB

Table 4: Hardware performance counters used to compare against model estimates

Value	Counter Name
Total Requests	MEM_LOAD_UOPS_RETIRED
L1 Misses	L2_RQSTS:ALL_DEMAND_REFERENCES
L2 Misses	LLC_REFERENCES
L3 Misses	LAST_LEVEL_CACHE_MISSES

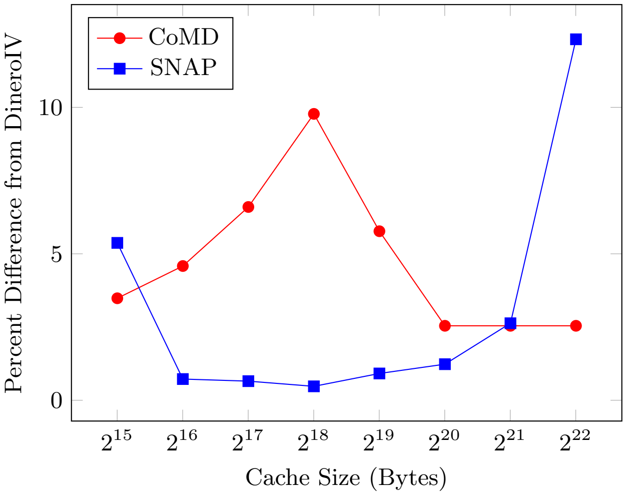


Figure 6: Model precision compared to the DineroIV cache simulator.

SNAP [27] is proxy application representing the computation, memory load, and communication behavior of PARTISN [7], a neutral particle transport application. It is written in Fortran and parallelized using OpenMP.

Traditional cache simulation forms the backbone of existing data movement analysis. We passed the IR-level address trace collected with Byfl into the DineroIV [19] cache simulator to compare with the absolute miss counts estimated by the stack model. The LLVM IR maintains its own address space, which is translated by the backend into actual memory addresses. Due to artifacts such as register spilling, additional operations may be introduced by this process, which is why a fair comparison necessitates invoking DineroIV on an LLVM-IR-level address trace. The purpose of our model is to project the performance of the IR-level addresses as a characterization of overall application data-movement behavior. Figure 6 shows the relative difference in miss counts for single-threaded

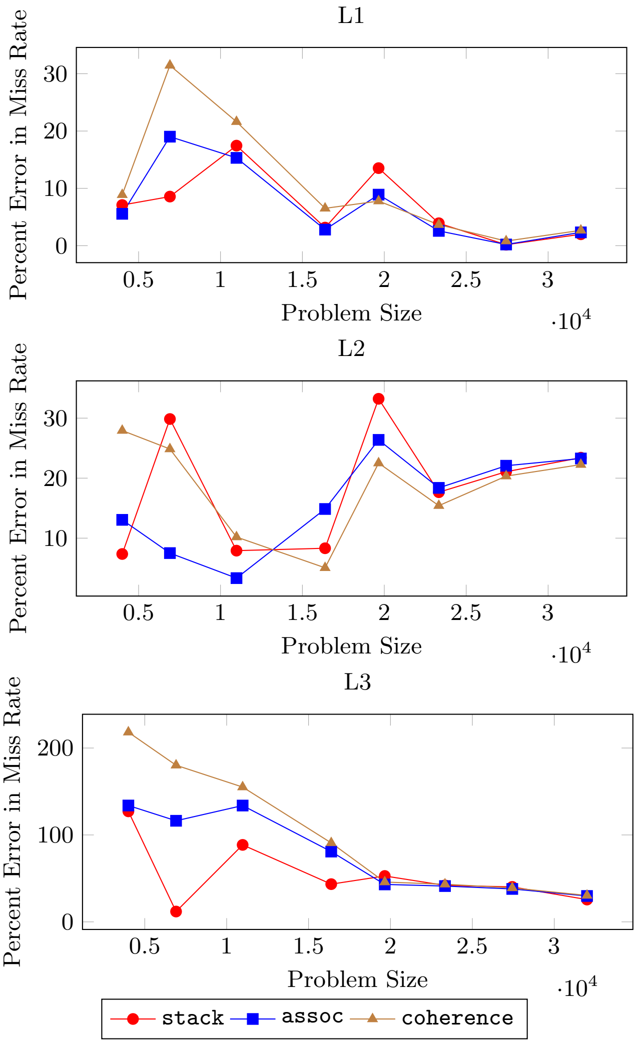


Figure 7: Model accuracy compared with hardware performance counters for each level of the cache hierarchy.

runs of both applications as a function of cache size. For this experiment, only a single pass of the stack model is made, achieving a precision of within 13% of DineroIV. This demonstrates that the stack model is capable of performing similarly to cache simulators when operating on the same address trace, with the added benefit of being able to delay selecting the depth of the cache hierarchy, the size of each cache, and the associativity until after the "simulation" has run.

To illustrate the accuracy of our stack model, we ran a multithreaded version of the application for multiple problem sizes, collecting performance counter values, to measure the actual miss rates exhibited by the node. We then ran the same applications, compiled with Byfl, generating miss rate estimates with the stack model for each level of the cache hierarchy. We compare three different versions of the model. The `stack` version only measures cache performance for the simple stack model, assuming full associativity and private cache space. The `assoc` model is augmented with the arbitrary associativity counter, and the `coherence` model includes both associativity counters and the effects of coherence traffic. Figure 7 shows the error in miss rates of these models over the hardware performance counters for CoMD is within 31% for the L1 cache and 33% for the L2 cache. The L3 cache performs to within 49% of the counters for larger problem sizes. We omit similar results for SNAP due to space constraints.

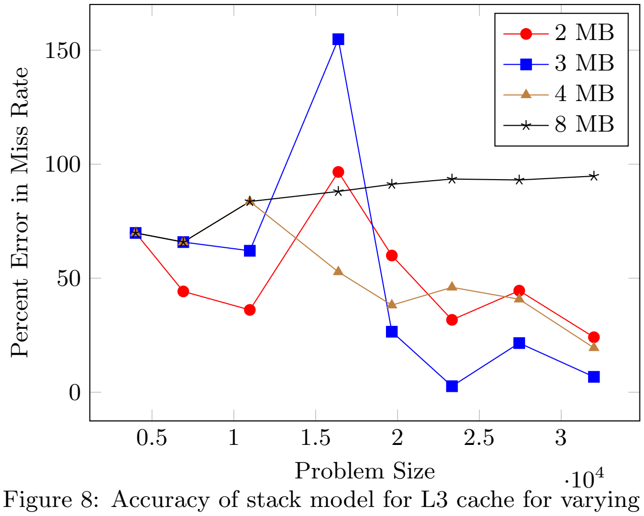


Figure 8: Accuracy of stack model for L3 cache for varying effective size.

Figure 7 illuminates an omission in the stack model. The architecture shares the L3 cache among all threads on the same core. The stack model naïvely subdivides the total shared stack space among all the cores. This results in a suboptimal allocation of resources, for two reasons. First, there is no guarantee that all threads will put equal demand on the L3 cache. It is likely that the thread with the highest demand for L3 cache resources will "win" the space from the other threads, giving the appearance of more *effective* cache space. Dynamic allocation policies in modern architectures follow strict criteria for ensuring fairness and performance [42]. Figure 8 shows how the miss-rate accuracy of the last level cache changes with the effective cache space per thread for runs of CoMD. The effective size of the cache given to each thread is not necessarily equal to an even allocation of the total shared space. The second reason for suboptimal allocation is that shared data in last level caches do not need to be duplicated; private cache partitions may contain duplicate data, as seen in Section 4.4. This also increases the effective cache space per thread as there is no longer a need to keep multiple copies of the same address.

6. CONCLUSIONS

This paper presented a technique for characterizing intranode data movement. Unlike traditional cache simulators, which simulate a single cache configuration in great detail, our approach gathers sufficient data during a single run of an application to predict miss rates of *any* depth of caches with *any* size and *any* associativity for each cache, all as a fast postprocessing step. Our technique is implemented as an LLVM compiler pass (as part of the Byfl analysis framework) that instruments an application to output a modicum of cache-access summary information—far smaller than a complete address trace—at run time. This information can then be used to predict cache rates for different cache configurations without needing to run the application.

Based on a set of experiments we performed comparing our stack model both to existing cache-simulation tools and to measurements taken on hardware performance counters, we conclude that we have developed an effective approach to rapidly analyzing how an application's memory-access pattern maps onto a large number of cache configurations. Instead of having to store a potentially large address trace to feed into multiple runs of a cache simulator or having to run an instrumented application repeatedly to simulate different cache configurations, we have shown that similar miss rates (within 13%) can be computed without large trace files and without long-running simulations of each cache hierarchy of interest. This work therefore has the potential to greatly improve the way that applications and cache hierarchies are designed and analyzed.

7. ACKNOWLEDGMENTS

The authors would like to thank Joshua Payne for his help with the shape-function example (Listing 1).

This work was supported by the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC52-06NA25396 with Los Alamos National Security, LLC.

8. REFERENCES

[1] `perf`: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`. [Online; accessed 8/29/2014].

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, Apr. 25, 2010. DOI: 10.1002/cpe.1553.

[3] G. Almási, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. In *Proceedings of the 2002 Workshop on Memory System Performance*, MSP '02, pages 37–43, New York, NY, USA, 2002. ACM. DOI: 10.1145/773146.773043.

[4] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*, pages 1–11, San Diego, California, USA, Jan. 10–13 1988. ACM. DOI: 10.1145/73560.73561.

[5] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, T. Mezzacappa, P. Moin, M. Norman, R. Rosner, V. Sarkar, A. Siegel, F. Streitz, A. White, and M. Wright. The opportunities and challenges of exascale computing—summary report of the advanced scientific computing advisory committee (ASCAC) subcommittee. *US Department of Energy Office of Science*, Fall 2010.

[6] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *Computer Architecture, 1988. Conference Proceedings. 15th Annual International Symposium on*, pages 73–80, May 1988. DOI: 10.1109/ISCA.1988.5212.

[7] R. S. Baker. A block adaptive mesh refinement algorithm for the neutral particle transport equation. *Nuclear Science and Engineering*, 141(1):1–12, May 15, 2002.

[8] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, Salt Lake City, Utah, USA, Nov. 10–16, 2012. DOI: 10.1109/SC.2012.71.

[9] L. A. Bélády. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. DOI: 10.1147/sj.52.0078.

[10] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000. DOI: 10.1177/109434200001400303.

[11] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000. DOI: 10.1177/109434200001400404.

[12] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[13] C. Caşcaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 150–159. ACM, 2003.

[14] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 340–351. IEEE, 2005.

[15] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55. DOI: 10.1109/99.660313.

[16] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *ACM SIGPLAN 2003 Conference on Programming language Design and Implementation*, pages 245–257, San Diego, California, USA, June 9–11, 2003. DOI: 10.1145/781131.781159.

[17] J. Dongarra and M. A. Heroux. Toward a new metric for ranking high performance computing systems. Sandia Report SAND2013-4744, Sandia National Laboratories, Albuquerque, New Mexico, USA and Livermore, California, USA, June 2013.

[18] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou. Experiences and lessons learned with a portable interface to hardware performance counters. In *International Parallel and Distributed Processing Symposium*, Nice, France, Apr. 22–26, 2003. IEEE. DOI: 10.1109/IPDPS.2003.1213517.

[19] J. Edler and M. D. Hill. Dinero IV trace-driven uniprocessor cache simulator. `https://pages.cs.wisc.edu/~markhill/DineroIV/`. [Online; accessed 8/29/2014].

[20] ExMatEx. CoMD proxy application. `http://www.exmatex.org/comd.html`. [Online; accessed 8/29/2014].

[21] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):703–746, 1999.

[22] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar'06)*, pages 1–9, Barcelona, Spain, Sept. 25–28, 2006. DOI: 10.1109/CLUSTR.2006.311904.

[23] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996. DOI: 10.1016/0167-8191(96)00024-5.

[24] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *Computers, IEEE Transactions on*, 38(12):1612–1630, 1989.

[25] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. Cmp $im: A pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, pages 28–36, 2008.

[26] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis transformation. In *2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2004)*, pages 75–86, San José, California, USA, Mar. 20–24, 2004. DOI: 10.1109/CGO.2004.1281665.

[27] Los Alamos National Lab. SNAP proxy application. `https://github.com/losalamos/SNAP`. [Online; accessed 8/29/2014].

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, Chicago, Illinois, USA, June 11–15, 2005. DOI: 10.1145/1065010.1065034.

[29] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970. DOI: 10.1147/sj.92.0078.

[30] J. Merrill. GENERIC and GIMPLE: A new tree representation for entire functions. In *GCC Developers Summit*, Ottowa, Ontario, Canada, May 25–27, 2003. URL: `http://ols.fedoraproject.org/GCC/Reprints-2003/GCC2003-Proceedings.pdf`.

[31] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.0*, Sept. 21, 2012. URL: `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`.

[32] S. Pakin and P. McCormick. Hardware-independent application characterization. In *2013 IEEE International Symposium on Workload Characterization (IISWC 2013)*, pages 111–112, Portland, Oregon, USA, Sept. 22–24, 2013. IEEE Computer Society. Extended abstract. DOI: 10.1109/IISWC.2013.6704676.

[33] D. A. Patterson and D. R. Ditzel. The case for the reduced instruction set computer. *SIGARCH Computer Architecture News*, 8(6):25–33, Oct. 1980. DOI: 10.1145/641914.641917.

[34] J. Payne, D. Knoll, A. McPherson, W. Taitano, L. Chacón, G. Chen, and S. Pakin. Computational co-design of a multiscale plasma application: A process and initial results. In *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2014)*, pages 1093–1102, Phoenix, Arizona, USA, May 19–23, 2014. IEEE. DOI: 10.1109/IPDPS.2014.114.

[35] D. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2–3 (June & September)):215–226, 2000. DOI: 10.1142/S0129626400000214.

[36] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*, pages 12–27. ACM, Jan. 10–13 1988. DOI: 10.1145/73560.73562.

[37] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT 2010)*, pages 53–64. ACM, 2010.

[38] J. Shalf, D. Quinlan, and C. Janssen. Rethinking hardware-software codesign for exascale systems. *IEEE Computer*, 44(11):22–30, Nov. 2011. DOI: 10.1109/MC.2011.300.

[39] Y. S. Shao and D. Brooks. ISA-independent workload characterization and its implications for specialized architectures. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 245–255, Austin, Texas, USA, Apr. 21–23, 2013. IEEE. DOI: 10.1109/ISPASS.2013.6557175.

[40] X. Shi, F. Su, J.-K. Peir, Y. Xia, and Z. Yang. Modeling and stack simulation of CMP cache capacity and accessibility. *Parallel and Distributed Systems, IEEE Transactions on*, 20(12):1752–1763, 2009.

[41] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th international conference on Supercomputing*, pages 1–12. ACM, 2001.

[42] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[43] J. Weidendorfer. Sequential performance analysis with Callgrind and KCachegrind. In M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *Tools for High Performance Computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, pages 93–113, Stuttgart, Germany, July 2008. Springer. DOI: 10.1007/978-3-540-68564-7_7.

[44] Y. Wu and R. Muntz. Stack evaluation of arbitrary set-associative multiprocessor caches. *IEEE Transactions on Parallel and Distributed Systems*, 6(9):930–942, 1995.