MIAMI: A Framework for Application Performance Diagnosis

Gabriel Marin
Innovative Computing Laboratory
University of Tennessee
gmarin@utk.edu

Jack Dongarra
Innovative Computing Laboratory
University of Tennessee
dongarra@utk.edu

Dan Terpstra
Innovative Computing Laboratory
University of Tennessee
terpstra@utk.edu

***Abstract*—A typical application tuning cycle repeats the following three steps in a loop: performance measurement, analysis of results, and code refactoring. While performance measurement is well covered by existing tools, analysis of results to understand the main sources of inefficiency and to identify opportunities for optimization is generally left to the user. Today's state of the art performance analysis tools use instrumentation or hardware counter sampling to measure the performance of interactions between code and the target architecture during execution. Such measurements are useful to identify hotspots in applications, places where execution time is spent or where cache misses are incurred. However, explanatory understanding of tuning opportunities requires a more detailed, mechanistic modeling approach. This paper presents MIAMI (Machine Independent Application Models for performance Insight), a set of tools for automatic performance diagnosis. MIAMI uses application characterization and models of target architectures to reason about an application's performance. MIAMI uses a modeling approach based on first-order principles to identify performance bottlenecks, pinpoint optimization opportunities, and compute bounds on the potential for improvement.**

I. Introduction

Investments in high performance computing (HPC) systems stand at tens of millions of dollars each year. These systems have tremendous peak performance potential, as demonstrated by their throughput results with highly optimized, dense linear algebra kernels [23]. However, most scientific simulations run at only a fraction of theoretical system peak speed. This large unfulfilled performance potential is due in part to compilers and application developers not being able to harness the potential of the architectures and in part due to an imbalance between the resources offered by current systems and the actual needs of applications. To close this performance gap, application developers must precisely understand what factors are limiting the performance of their codes, a process known as *performance diagnosis*. Performance diagnosis is the first step, and at the same time, the most difficult step of any performance optimization effort, just as understanding the causes behind a program crashing or producing incorrect results is the most important and the most difficult step of any program debugging effort. Once we identify the factors that limit performance, the code transformations required to alleviate the detected performance bottlenecks become more easily apparent.

State of the art performance analysis tools in use today use either caliper-based hardware counter measurements [3], [21] or hardware counter sampling [1], [10], [20] to measure application performance during execution. A strength of hardware performance counters is that they can observe phenomena that cannot be measured directly otherwise. However, hardware counters can only observe performance effects, the result of interactions between code and target architecture. A process of deconvolution through which we can attribute parts of the observed effects to specific application and architectural factors is needed to perform root cause analysis from hardware counter measurements. While certain correlations between application or architectural factors and the observed performance effects can be established, the process requires high levels of user expertise and a significant amount of guesswork.

To provide the kind of feedback that we think is necessary, tools must identify and model in isolation the application and architectural factors that are important for performance, *e.g.* the application instruction mix, the instruction schedule dependencies, the type of resources available on a target architecture and the type of resources required by each basic operation during execution, they must understand how data is reused and the patterns with which an application traverses memory. Performance diagnosis tools must then use a performance convolution process based on first-order principles to understand the factors that are limiting performance at each point in an application. An estimate of the maximum potential for improvement can be computed by idealizing the limiting factors and reapplying the convolution process. Finally, understanding what factors are limiting performance, directly determines the type of code or architectural changes that are needed to alleviate that bottleneck. In some instances, such transformations are not possible, or they are prohibitively expensive. However, it is very useful for a user to identify such situations, so as to understand when to stop optimizing. Providing users with an accurate trade-off of costs, *i.e.* the type of transformations that are required, and benefits, *e.g.* the potential for performance gains, enables users to make informed decisions about where to focus their tuning efforts.

To be useful, tools must automate as much of this process as possible. They must work on full applications instead of requiring users to extract "interesting kernels," and they must be able to handle interactions between application code and system libraries. Because performance depends also on the quality of the code produced by the compiler, tools should try to observe the effect of optimizations while not perturbing the optimization process. For these reasons, we think that the best way to perform performance diagnosis is by analyzing application executables. In addition, tools that work on binaries can naturally handle applications written in different programming languages or using different programming models.

In this paper, we present MIAMI (**M**achine **I**ndependent **A**pplication **M**odels for performance **I**nsight), a set of extensible tools for automatic performance diagnosis. MIAMI analyzes fully optimized x86 application binaries to construct a machine-independent understanding of an application's algorithm and implementation. It uses a detailed model of a target architecture, to reason about the thread performance of an application on that architecture. MIAMI uses a modeling approach based on first-order principles to understand performance inefficiencies, to identify the performance limiting factors inside each loop, and to provide insight into the code transformations needed to alleviate them.

The rest of this paper is organized as follows. Section II presents our motivation and design goals with developing MIAMI. Section III describes the implementation of the MIAMI framework. Section IV describes three case studies that illustrate the type of performance insights provided by MIAMI. Section V reviews existing performance analysis techniques and contrasts them against our approach. Section VI summarizes our findings and concludes the paper.

II. Motivation and Design Goals

While node concurrency will continue to increase for many years, fundamental architectural changes are needed at the node level to achieve the levels of energy efficiency required for an exascale machine [12]. Future systems will consist of increasingly complex architectures with massive numbers of potentially heterogeneous components [8], longer vector units, and deeper memory hierarchies. Meanwhile, hierarchies of large, multifaceted software components will be required to build next-generation applications.

Understanding how to take advantage of new architectures, identifying where performance is lost and what is causing it, while also important in the past, will become even more important in the future as we face a more diverse architectural landscape. Fundamental issues such as automatic root cause analysis of performance bottlenecks, and automatic recognition of optimization opportunities are not fully solved problems yet.

We do not think that any single approach can explain all performance inefficiencies. Tracing tools work best for understanding load imbalances in parallel applications or for diagnosing inefficiencies caused by the timing of communication events, such as late receives. Hardware counter measurements have no equal when it comes to understanding phenomena that is micro-architecture specific, such as bank conflicts, misaligned accesses, invalidation of modified cache lines, remote memory accesses, or miss events in the processor's front-end. Some of those phenomena could be otherwise understood only by cycle accurate simulations, but such simulators are very difficult to develop and much more expensive to run.

However, hardware counters are deficient at explaining performance inefficiencies in the CPU's back-end. Metrics such as the number of instructions retired per cycle (IPC) provide a measure of the distance from machine peak retirement rate. However, if the measured IPC is low, we cannot easily pinpoint the factors that are limiting performance. We can try to guess and then try to validate our assumptions, using a trial and error approach. Memory delays are always a good scapegoat for explaining low performance. However, many times applications run inefficiently even when data is read from the L1 cache. One of our goals with MIAMI is to provide insight into inefficiencies in the processor's back-end.

Many performance analysts in the HPC community judge the speed of a code by the rate at which floating point instructions are issued, and the code's efficiency by how close its floating-point issue rate is to the machine's theoretical peak FLOPS performance. However, unlike highly optimized dense linear algebra kernels, instruction mixes found in typical scientific applications have large fractions of register copy, data shuffling, integer arithmetic, branches and memory instructions [24]. All these instructions compete for the limited CPU retirement bandwidth when the ILP is high, resulting in a low floating point issue rate on average. Other times, there is an imbalance between the type of floating point instructions in an application loop and the mix of floating point execution units on the target machine. Modern architectures usually have a balanced number of `Add` and `Multiply` execution units. If a calculation uses only one type of instructions, for example only additions, it can achieve no better than 50% of theoretical machine peak FLOPS rate, before any other sources of inefficiencies are even considered. We found our analysis equally useful for explaining why an application or an application loop does not run as fast as we think it should due to an inherent characteristic of the algorithm or of the implementation. Even if there are no obvious code transformations to accelerate the computation in such cases, it is very useful to understand that fact.

MIAMI uses static analysis of application binaries to understand the instruction schedule dependencies and the precise instruction mixes inside loop bodies. It uses this information to pinpoint cases when the code is limited by dependency cycles, by skewed instruction mixes, or if the code does not take advantage of available vector units.

An even more significant source of performance losses is poor data locality. Traditional tools measure and report the number of cache misses inside individual loops and routines. Knowing which loops of an application experience a large number of cache misses does not provide sufficient insight by itself for understanding how to improve data locality, because data reuse is not a local phenomenon. The same data structures can be accessed by multiple loops located in different routines. MIAMI uses static and dynamic analysis on application binaries to identify the memory reuse patterns that create the largest number of cache misses. The information reported about a memory reuse pattern directly determines the type of loop transformations needed to improve its locality.

Contemporary architectures include also one or several streaming hardware prefetchers. Even if not all data transfers from memory can be eliminated through data locality optimizations, we can still improve application performance by tuning the memory access patterns for the hardware prefetchers. MIAMI helps this process by identifying the memory accesses that cannot be effectively prefetched and the reasons why.

While working on application binaries provides programming language and programming model independence, it also ties our analysis framework to one architecture family. For this reason, one of our design goals with MIAMI was to define a program intermediate representation, called the *MIAMI IR*, as seen in Figure 1. The MIAMI IR isolates the bulk of our analysis performed in the tool's back-end, from the native architecture and the selected binary instrumentation framework. We think that this design choice will ease the tools' maintainability and eventual portability to new architectures or binary formats.

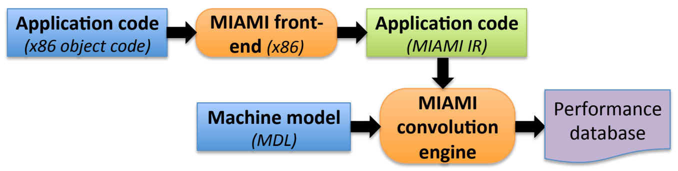


Fig. 1. The MIAMI workflow

III. MIAMI Implementation

Figure 1 shows a high level diagram of the MIAMI workflow. MIAMI makes the following contributions:

MIAMI intermediate representation – MIAMI uses a machine-independent intermediate representation (IR) of applications that includes control flow information as well as a machine-independent representation of computation and data flow inside each basic block. Using a machine-independent IR enables the performance convolution back-end to reason about an application's performance across different architectures.

Machine description language – MIAMI uses a machine description language (MDL) for modeling target architectures. The MDL constructs are tightly coupled with the machine-independent application IR.

A front-end for x86-64 binaries – MIAMI uses static and dynamic analysis of application executables and shared libraries to automatically build a machine-independent representation of full applications. Currently, the tool includes an x86 front-end to decode x86 binaries into the MIAMI IR.

Performance convolution back-end – MIAMI uses a performance convolution engine that combines dynamic and static information about an application together with a target architecture model to reason about application performance. The convolution engine uses first-order principles to map application operations onto the target architecture resources to compute performance estimates, identify performance bottlenecks and determine the potential for performance improvement.

Data locality analyzer – Data movement inside the memory hierarchy is typically the main cause of performance losses for scientific applications, and one of the principal sources of energy consumption in current systems. The data locality analyzer identifies memory access patterns with poor cache locality and patterns that are prefetcher unfriendly.

A. The MIAMI Intermediate Representation

Our x86 front-end uses PIN [16] to instrument object code and to perform dynamic analysis on application binaries, and XED [11] to decode the instructions from an x86 binary into their MIAMI representation. To increase the tools' portability to eventually other binary formats and architectures, we defined a MIAMI intermediate representation and our analysis is performed on the MIAMI IR. The MIAMI IR defines its own abstractions for load modules and routines, and captures information about the application control flow and about the computation inside each basic block. A load module corresponds to one object code file. It can be a static or a dynamically linked executable, or a shared library. We build a control flow graph (CFG) for each routine, and we decode the machine instructions from the native binary into their generic MIAMI representation that resembles RISC instructions. Thus, arithmetic x86 instructions with memory operands are decoded into multiple micro-ops: one for the actual arithmetic operation, plus one additional micro-op for each memory read and write operation performed by the instruction. We associate a list of MIAMI instructions with each basic block. We store detailed information about a MIAMI instruction. This information is sufficient to perform dependence analysis and data flow analysis on the MIAMI IR.

```
CpuUnits = U_ALU*3, U_Load*2, U_StAddr*2,
           U_STD, U_VMul, U_VAdd, U_JMP,
           U_VShuffle*2, U_FDiv, U_FpAdd,
           U_FpMul, U_FpShuf, U_FpBool,
           U_FpBlend*2, U_Carry, I_Port*6;

AsyncResources = L1_RdTrans*2, L1_WrTrans;
```

Fig. 2. Machine resources defined for an Intel Sandy Bridge

B. The Machine Description Language

We use a machine description language (MDL) to build models of target architectures. A machine model enumerates the resources available on the target architecture, describes execution templates for each generic micro-operation type of the IR, specifies eventual use conflicts among different resources, allows for optional idiom replacement rules to account for differences in the instruction set architecture (ISA) of different machines, provides information about the memory hierarchy, and can be used to specify other characteristics of the target architecture.

A machine model includes detailed information about the resources needed and the scheduling constraints of each MIAMI instruction type. We use information from vendor documents and from micro-benchmarks to build such a model. A machine model is constructed by hand, but we only have to do this task once for each architecture. A good performance analyst must have a general familiarity with computer architecture, and should have good knowledge about the target architecture at hand. MIAMI captures that architecture knowledge in the machine model, which can be written by experts and then made available to the community. The tools users are not burdened with knowing low-level architecture details.

A machine model starts by defining the machine resources that can constrain the scheduling of instructions, such as the set of execution units and the machine issue ports. Figure 2 shows the set of CPU resources defined for an Intel Sandy Bridge processor, based on the processor diagram provided in vendor documentation [9]. For each MIAMI IR instruction class, the model must define at least one execution template that describes the resources used by such an instruction during execution. A template defines the resources used over a number of cycles. The default latency of an instruction is determined by the length of its shortest execution template. The resources used in various cycles create scheduling constraints with other instructions. Figure 3 shows sample execution templates for a few different flavors of `Load` and `Store` micro-operations on a Sandy Bridge machine. The `NOTHING` keyword is used to indicate that no scheduling constraints can occur in those cycles because the machine units are pipelined, but they help define the instruction's latency.

```
Instruction Load:fp,vec{128} template = I_Port[2]+U_StAddr[0], U_Load, NOTHING*3 |
                                        I_Port[3]+U_StAddr[1], U_Load, NOTHING*3;
Instruction Load:fp,vec{256} template = I_Port[2]+U_StAddr[0],U_Load,U_Load,NOTHING*4 |
                                        I_Port[3]+U_StAddr[1],U_Load,U_Load,NOTHING*4;

Instruction Store:fp,vec{128} [L1_WrTrans] template =
    I_Port[2]+U_StAddr[0], I_Port[4]+U_STD | I_Port[3]+U_StAddr[1], I_Port[4]+U_STD;
Instruction Store:fp,vec{256} [L1_WrTrans(2)] template =
    I_Port[2]+U_StAddr[0], I_Port[4]+U_STD | I_Port[3]+U_StAddr[1], I_Port[4]+U_STD;
```

Fig. 3. Sample `Load` and `Store` execution templates for an Intel Sandy Bridge architecture

Optionally, the machine model can define a set of asynchronous resources. These resources are handled differently by the MIAMI evaluation engine. Instruction templates can indicate the use of asynchronous resources with an optional quantity amount. Asynchronous resources do not impose scheduling constraints on individual instructions. Instead, their cumulative use defines a lower bound on the schedule length for an entire loop, in other words, a loop throughput limit.

The Sandy Bridge processor has a non-trivial memory subsystem. It has two ports for issuing `Load` requests, and one port dedicated to `Stores`. However, it has only two address generation units (`U_StAddr` in Figure 2), that must be shared by any `Load` and `Store` instructions. Therefore, it can sustain either two `Loads`, or one `Load` and one `Store` every cycle. Moreover, its bandwidth to the L1 cache is limited to two 16 byte read transfers, and one 16 byte write transfer every cycle. Thus, an AVX workload is effectively limited to a throughput of one 256-bit `Load` every cycle, and one 256-bit `Store` every two cycles. The templates in Figure 3 try to capture all these constraints. The rule for 256-bit `Loads` uses the `U_Load` unit for two cycles, effectively halving the instruction's throughput vs. its 128-bit counterpart. The rule for 256-bit `Stores` makes use of two units of the asynchronous resource `L1_WrTrans`. The intuition here is that while a constraint on the load bandwidth can potentially increase the latency of a `Load`, and thus delay any instructions that depend on the read data, `Stores` write data to a store buffer that is transferred asynchronously to the memory subsystem, and therefore, affect only the loop throughput. Many other instruction classes have generally simpler execution templates.

The MDL provides also constructs to specify special bypass latencies based on the producer / consumer instructions, and based on the dependence type. In addition to register dependencies, the MIAMI dependence analyzer also uncovers memory dependencies and it creates control dependencies to restrict the ordering of instructions around function calls and inner loops. Thus, most control dependencies are defined to have a latency of zero cycles, and memory dependencies are defined to have a latency of one cycle, because `Loads` can generally be issued one cycle after a `Store` to the same address, data being forwarded from the store buffer.

Another important language construct is the *replacement rule*. Replacement rules specify that certain instruction idioms should be replaced with different instruction patterns, and are useful to account for differences in the instruction set architecture (ISA) of different machines, or to define special micro-architecture optimizations. An example of the former use case would be to construct a machine model for an AMD Bulldozer or an Intel Haswell processor, knowing that the application could have been compiled for an earlier architecture and is not using the new fused multiply add instructions. We can write a replacement rule that finds `Add` instructions as sole consumers of `Mult` instructions, and replaces them with `MulAdd` instructions. As an example of the latter use case, the replacement rules in Figure 4 specify that `Sub` and `Xor` instructions with identical source operands should be converted to `NOPs` without source operands. Intel calls such instructions *dependency breaking idioms* [9], and they are recognized and removed by a CPU's front-end.

```
Replace Sub:int $rX, $rX -> $rY
   with NOP:int -> $rY;
Replace Xor:int $rX, $rX -> $rY
   with NOP:int -> $rY;
```

Fig. 4. Dependency breaking idioms defined using replacement rules.

C. Diagnosing Computational Inefficiencies

MIAMI uses a modulo instruction scheduler to map an application's instructions onto the resources of a target architecture. This convolution process identifies performance inefficiencies due to an interaction between application code and a target machine's back-end resources. For this, MIAMI uses a two step process. In the first step, a profiler collects CFG edge execution frequencies at run-time. This step adds overhead proportional to the application running time, between 50% and 200% for most applications. Most of the analysis is performed offline in a second step. The second step's overhead is proportional to the size of the application code. MIAMI uses the collected profile information and static analysis on the application binary to identify the executed paths in application loops and their execution frequencies. MIAMI reschedules these paths for a target architecture using a critical-path driven, modulo instruction scheduler that has been instantiated with a model of the architecture.

MIAMI computes instruction schedules one path at a time. Scheduling each path separately emulates an ideal branch predictor. The computed instruction schedule cost, C_{actual}, takes into account both the instruction schedule dependencies and the machine resource constraints. MIAMI reports the predicted execution times at loop and path level. In addition, it computes *cycle accounting* – it provides a detailed breakdown of the factors contributing to the execution time, such as application dependencies or contention on various machine resources. It also performs *resource use accounting* – it reports a summary of how many cycles each machine resource has been in use or idle inside each loop, and a precise breakdown of the instruction mix at loop level, including operation types, operand widths, and vector lengths.

The metrics computed by MIAMI provide insight into the factors that are bounding performance inside each loop. Many times, the relationship between these metrics is sufficient to diagnose the sources of inefficiency. However, interpreting that information and understanding where the low-hanging opportunities for improvement are, still requires experienced users with a background in compilers or architectures.

To make the process of uncovering opportunities for optimization easier, we also report a few higher level metrics. By idealizing one or several constraints limiting execution time, we can compute a lower bound on the execution cost, C_{ideal}, that can be achieved if we can relax or remove said constraints. The maximum potential for improvement from removing the respective scheduling constraints can be simply computed as the difference between C_{actual} and C_{ideal}.

We compute the maximum potential for improvement from **increased ILP**, from **additional machine resources**, and from **ideal vectorization**, and we report these metrics at the loop and the routine level. The **improvement from increased ILP** metric is computed by relaxing all data dependencies while preserving instruction mixes and control dependencies. The **improvement from additional machine resources** metric preserves instruction mixes and all instruction dependencies, but assumes the machine has unlimited resources.

Efficient use of SIMD execution units is essential for achieving a significant fraction of peak performance on current and future systems. Therefore, it is very important to identify loops that can benefit from vectorization, and estimate the performance gains from doing so. The **improvement from ideal vectorization** metric relaxes data dependencies, but it also looks at the instruction mixes and at the machine model. This “what if” analysis packs together instructions for which the machine model provides templates of longer vector lengths, by unrolling a loop the required number of times. For example, we have to unroll a loop four times to pack scalar double precision arithmetic into AVX vectors. Instructions already vectorized and instructions that do not have a corresponding template of longer vector size are duplicated when the loop is unrolled. Instructions used for address arithmetic and for the loop branch condition are not vectorized, but they are neither duplicated when the loop is unrolled. Our goal is to provide a sensible bound on the potential gains from vectorization. This metric should provide a potential for improvement at least as large as that from increased ILP, since we are relaxing instruction dependencies in addition to looking for vectorization opportunities. Thus, when we look at these metrics, we need to consider the additional improvement provided by vectorization over the improvement from increased ILP.

D. Diagnosing and Improving Data Locality

A large amount of literature for compiler research shows that program transformations can dramatically reduce the number of cache misses and improve program performance through high level loop transformations such as loop interchange, unroll-and-jam [7], cache blocking (tiling) [26], loop fusion [14], or a combination of transformations. Identifying that memory bandwidth or memory latency is limiting application performance, represents just a preliminary step. Even knowing which loops of an application experience a large number of cache misses, usually does not provide sufficient insight by itself for understanding how to improve locality. The reason for this is that data reuse, the main determinant of cache performance, is not a local phenomenon. The same data may be accessed in multiple loops located in different routines. Given a source-code data array that experiences long temporal reuses, two crucial pieces of information needed to understand how to restructure the code to shorten its reuse are to identify the place where the array has been previously accessed, and the application code executed between data reuses.

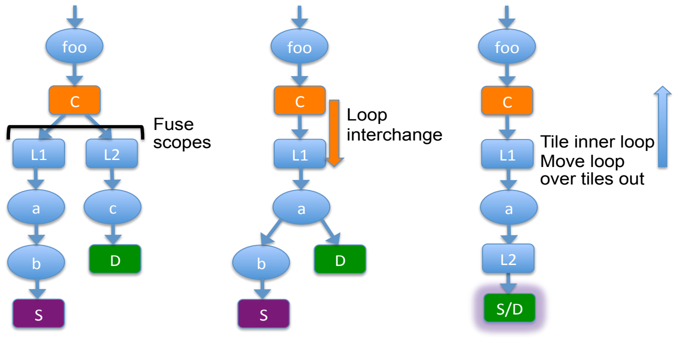


Fig. 5. Examples of source, destination and carrier scope relationships

We use reuse distance analysis to understand the temporal locality of each memory access. To understand how to improve data locality, in other words, how to shorten a data reuse, we must also understand where those intervening accesses are located. SLO [2] obtains this information by collecting the set of basic blocks executed between two accesses. In previous work on Sparc binaries [18], we captured the same information by understanding how far back the flow of control returns in the dynamic program calling context tree between consecutive reuses of a datum. MIAMI implements the latter approach. We maintain information about the dynamic calling context tree during execution. In our implementation, a dynamic calling context tree path includes both the routine frames and the loop nests encountered from an executing instruction to the root of the tree. We say that the *carrier scope* of a data reuse is the shallowest program scope where the control flow returns between two consecutive access to the same datum.

MIAMI uses dynamic analysis to identify the memory reuse patterns in an application during execution and to associate a histogram of observed memory reuse distances with each pattern. A memory reuse pattern is identified by the following information: the carrier scope, the path from the carrier scope to the first access to the datum, also called the source of the reuse, and the path from the carrier scope to the second access to the datum, which we call the destination of the reuse. In an offline step, the MIAMI back-end translates the measured memory reuse distances into a prediction of cache misses, based on the memory hierarchy parameters specified in the machine model. MIAMI identifies and ranks the memory reuse patterns that produce the highest number of cache and TLB misses for a particular target machine.

The relationship between the three scopes associated with a reuse pattern directly determines the type of code transformations needed to improve locality. Figure 5 shows a few examples. In all three scenarios, the orange `C` loop is the carrier scope, the `S` block is the source of the reuse, and the `D` block is the destination of the reuse. When a reuse is carried within the same iteration of the carrier scope, we must fuse the scopes on the paths to the source and destination scopes, starting from the carrier scope. When the reuse is carried across iterations, we must interchange the carrier scope into an inner position, or we must tile one of the inner loops and move the loop over tiles outside the carrier scope. The intuition in all these cases is that we are reducing the amount of other data touched between data reuses. MIAMI can identify the transformations required to shorten a reuse. However, such transformations are not always legal or easy to implement. In general, the farther removed the carrier scope is from the places where data is accessed, the harder it is to improve the reuse, because there is a greater chance of data dependencies that prevent refactoring.

Understanding bandwidth constraints: The reuse distance models provide a lower bound on the traffic volume between any two adjacent levels of the memory hierarchy. For each memory level, MIAMI determines the minimum time needed to transfer the data from the next memory level, using the peak bandwidth values from the machine model, and assuming ideal prefetching. It then compares these times to the estimated instruction schedule cost for a loop to understand if the loop is potentially bandwidth constrained. This metric is similar to the loop balance metric introduced in [6], except that we compute the balance between memory bandwidth and the effective instruction schedule cost of a loop, not just its floating-point peak performance.

E. Tuning for the Hardware Prefetchers

We cannot optimize away all transfers of data from memory for a typical application. However, we can still improve an application's performance by tuning its memory access patterns for the hardware prefetchers. Modern architectures include one or several streaming hardware prefetchers that can hide part of the memory latency when applications traverse memory with the right access patterns. MIAMI uses dynamic analysis to abstractly understand streaming behavior in applications. It uses static analysis to pinpoint memory accesses that cannot be effectively prefetched and to identify the reasons why, such as non-streaming accesses, streaming accesses with a too large stride, or loops with too many concurrent streams. A detailed description of this analysis is provided in [17].

F. Support for Parallel Applications

While application scalability is increasingly important, efficient utilization of resources on each core is crucial for both performance and power efficiency. A large number of tools support collection of traces for parallel applications to diagnose load imbalances. While MIAMI focuses on providing performance diagnosis insight for individual threads, it also supports data collection and analysis for parallel applications. The MIAMI profilers support both MPI and multithreaded applications. The tools automatically collect and save profile data separately for each application thread. The profile data collected for any thread can be then post-processed by the MIAMI back-end to understand performance inefficiencies.

G. Analysis of Results

MIAMI outputs performance results in CSV format, to enable parsing with a post-processing script and visual analysis using a spreadsheet application, and in XML format for top-down analysis using `hpcviewer`, the viewer application distributed with HPCToolkit [1]. In this paper, we include screenshots of MIAMI results displayed in `hpcviewer`.

```
void compute(int reps) {
  int i, j, k, r;
  for(r=0 ; r<reps ; ++r)
    for(i = 0; i < N; ++i)
      for(j = 0; j < N; ++j)
        for(k = 0; k < N; ++k)
          C(i,j)+=A(i,k)*B(k,j);
}
```

Fig. 6. Matrix multiply code

The viewer presents data in tabular format. Each metric is shown in a separate column. The rows correspond to program scopes such as loops and routines. We can expand a particular scope to understand the contribution of its inner scopes to the various metrics. The percentages visible on the right side of each cell (see Figure 8) are added by the viewer and are computed column-wise. They represent the contribution of a scope (and its children) to the total value of that metric corresponding to the entire experiment. The percentage is useful to quickly assess that a particular routine is responsible for 25% of the running time of an application, or that it accounts for 40% of the potential for improvement. However, many times we are interested in the relationship between different metrics for a particular scope. In such cases, we have to ignore the pre-computed percentages and compare the absolute values displayed inside the cells on the same row.

IV. Case Studies

In this section, we describe our experiences with applying MIAMI to three applications, a simple matrix multiply algorithm, a PLASMA [5] math kernel, and the LULESH proxy application [15]. Our motivation with these case studies is to demonstrate some of the performance diagnosis insight provided by MIAMI, and to show how this insight can be used to find low-hanging tuning opportunities. All experiments were performed on a system with dual Intel Xeon E5-2690 CPUs, based on the Sandy Bridge micro-architecture. Each processor has 8 cores and a shared 20 MB L3 cache.

A. Matrix Multiply

Our first experiment looks at the matrix multiply code shown in Figure 6. We added a fourth outer loop that repeats the matrix multiplication several times, to keep the number of calculations within the same order of magnitude as we vary the matrix size. The number of repetitions is computed as $\lfloor \frac{1024^3}{N^3} \rfloor$. We compiled this code with two compilers. First binary was compiled with the GNU compiler version *4.7.1* and flags `-Ofast -g`. The second binary was compiled with the Intel compiler version *13.1.2* and compilation flags `-O3 -xHost -g`. Our goal here is not to compare the performances of the two compilers, because we did not attempt to find the best codes these compilers can produce. Our goal is to get two different binaries, and use MIAMI to understand the factors behind the obtained execution times.

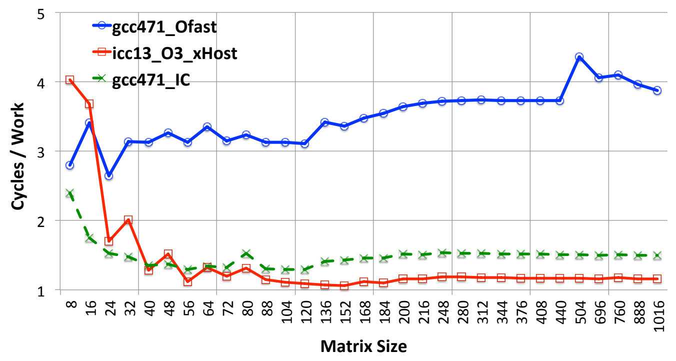


Fig. 7. Matrix multiply: time per unit of work using two compilers

Curves **gcc471_Ofast** and **icc13_O3_xHost** in Figure 7 plot the average time per unit of work for the two binaries running on our test system. We divided the total execution time, including initialization code, by the amount of useful work, *i.e.* $reps{*}N^3$. Lower is better for this type of graph. The times were measured using hardware performance counters.

We notice that the codes produced by the two compilers have vastly different performances. In fact, at problem size `120`, the second binary runs almost three times as fast as the first binary, 1.08 vs. 3.11 cycles/work. These results underline the importance of working at binary level to observe the effects of compiler optimizations. If we were presented only with the performance results of the first binary, and if this was not such a well known code for which we know that a highly tuned implementation can achieve performance close to machine peak FLOPS, we would have a hard time knowing if the performance of the first binary was good or bad. Moreover, the first binary has an average IPC of 2.6 at problem size `120`, which is generally considered excellent. Note that a Sandy Bridge core can retire at most 4 instructions per cycle.

We used MIAMI to analyze the two matrix multiply binaries using matrices of size `120`. Figure 8 shows a screenshot of several performance metrics computed by MIAMI for the binaries compiled using the GNU compiler (top), and the Intel compiler (bottom). The first five metrics shown in Figure 8 for the first binary are, in order: execution time, improvement potential from additional machine resources, improvement potential from additional ILP, improvement potential from vectorization, time due to application dependencies. The highlighted line corresponds to the cyclic path through the innermost loop. MIAMI shows values per iteration for a path, while for a loop scope it shows metrics aggregated over all the paths and all the iterations. The cyclic path inside the inner loop is limited by a recurrence of `3` cycles (metric `AppDepTime`), which is the latency of a floating point addition on a Sandy Bridge. The matrix multiply code has a reduction in the inner loop.

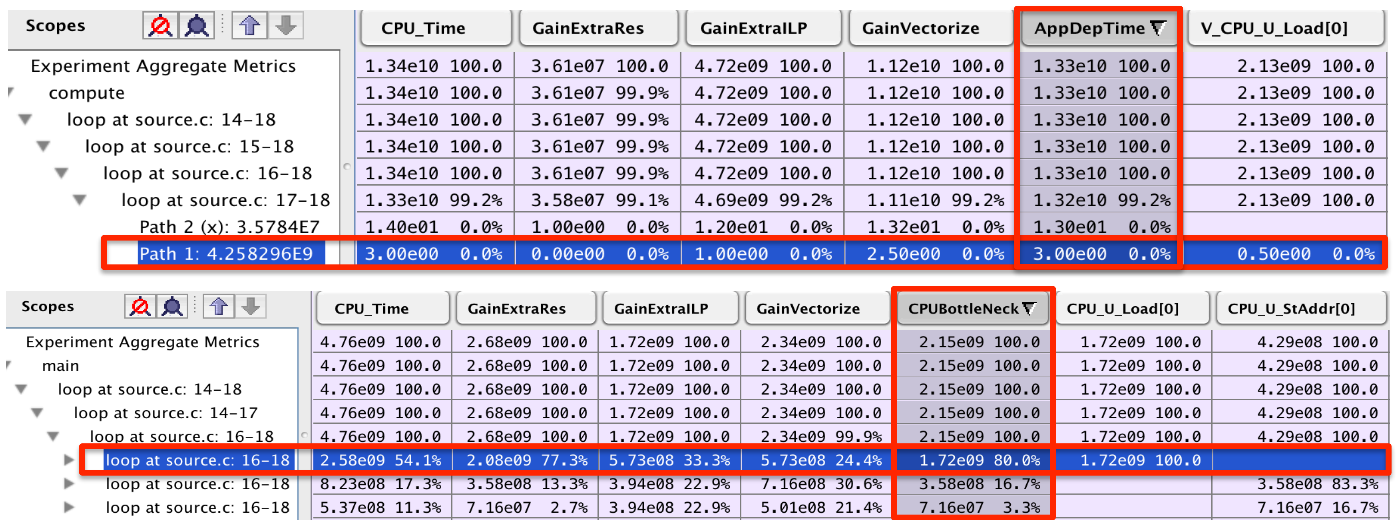
Obviously, the improvement potential from additional machine resources is very low, *3.61e07* cycles out of *1.34e10* total execution time, due to the recurrence in the inner loop. On the other hand, removing data dependencies would cut *1* cycle (metric `GainExtraILP`) from the cost of the cyclic path through the inner loop, and roughly 33% of the total execution time. The improvement is modest because the compiler generated many auxiliary instructions for address arithmetic. The code already has a good amount of ILP. The potential for improvement from vectorization (metric `GainVectorize`) is significantly higher. The average cost of one inner loop iteration would drop from `3` to `0.5` cycles. This is much better than the improvement from additional ILP. These metrics suggest that the code is not using vector operations, an observation confirmed by looking at the instruction mix reported by MIAMI for the innermost loop. The last metric displayed in the top table of Figure 8 shows that if the code was perfectly vectorized, it would be limited by contention on the `U_Load` units. In our Sandy Bridge machine model, see Section III-B, these units model the download bandwidth from the L1 cache.

The second binary has an average IPC of 2, but runs three times as fast as the first binary. The first four MIAMI metrics displayed for this binary are the same as for the first binary. The last three metrics show the total time due to resource contention, the time due to contention on the `U_Load` units, and the time due to contention on the `U_StAddr` units. We notice that the object code has three level four loops. The Intel compiler applied loop interchange and unrolled the inner loop, processing 16 elements in one iteration. Two of the level four loops handle the odd iterations from unrolling. The unrolled inner loop, highlighted in the figure, is limited primarily by contention on the `U_Load` units, *1.72e09* out of *2.58e09* cycles. The other two loops are partially limited by contention on the address generation units, `U_StAddr`, which indicates a too high ratio of memory to arithmetic instructions.

For the main level 4 loop, the improvement potentials from additional ILP and from vectorization are equal, which suggests the code is already vectorized. The instruction mix report confirms this observation. Most arithmetic and memory instructions are 256-bit, the exception being some `load` instructions that are 128-bit wide. However, packing these instructions into AVX `loads` would not speed-up the code. The bottleneck remains the load bandwidth from the L1 cache. While the Intel compiler generated well vectorized code for the unrolled loop, the loop runs at only 25% of peak, due to contention on the L1 bandwidth. To achieve higher throughput, the code must increase data reuse in registers.

While improving register reuse is non-trivial, we attempt here to improve the performance of the code produced by the GNU compiler based on the insight provided by MIAMI. The recurrence in the inner loop is the main bottleneck for the first binary. The `-Ofast` flag already asks the compiler for vectorization. However, the GNU compiler fails to either interchange the loops or to apply unroll&jam. Therefore, we manually interchanged loops 3 & 4 and recompiled the code with the GNU compiler. The performance of this code is shown in Figure 7 under the label **gcc471_IC**. The GNU compiler generated fully vectorized arithmetic operations. However, the code uses only memory operations that operate on half an AVX vector at a time. As a result, this binary is limited by contention on the address generation units, `U_StAddr`, and runs slightly slower than the binary produced by the Intel compiler.

Our goal with MIAMI is to provide performance insight, not to have perfect predictions of execution time. However, having accurate execution time estimates when the data fits in cache, increases confidence in our approach and in our machine model. The execution times measured with hardware counters for the two binaries are $1.33e10$ and $4.68e09$, respectively, which are within 2% of the times estimated by MIAMI.



Scopes	CPU_Time	GainExtraRes	GainExtraILP	GainVectorize	AppDepTime	V_CPU_U_Load[0]
Experiment Aggregate Metrics	1.34e10 100.0	3.61e07 100.0	4.72e09 100.0	1.12e10 100.0	1.33e10 100.0	2.13e09 100.0
compute	1.34e10 100.0	3.61e07 99.9%	4.72e09 100.0	1.12e10 100.0	1.33e10 100.0	2.13e09 100.0
loop at source.c: 14–18	1.34e10 100.0	3.61e07 99.9%	4.72e09 100.0	1.12e10 100.0	1.33e10 100.0	2.13e09 100.0
loop at source.c: 15–18	1.34e10 100.0	3.61e07 99.9%	4.72e09 100.0	1.12e10 100.0	1.33e10 100.0	2.13e09 100.0
loop at source.c: 16–18	1.34e10 100.0	3.61e07 99.9%	4.72e09 100.0	1.12e10 100.0	1.33e10 100.0	2.13e09 100.0
loop at source.c: 17–18	1.33e10 99.2%	3.58e07 99.1%	4.69e09 99.2%	1.11e10 99.2%	1.32e10 99.2%	2.13e09 100.0
Path 2 (x): 3.5784E7	1.40e01 0.0%	1.00e00 0.0%	1.20e01 0.0%	1.32e01 0.0%	1.30e01 0.0%	
Path 1: 4.258296E9	3.00e00 0.0%	0.00e00 0.0%	1.00e00 0.0%	2.50e00 0.0%	3.00e00 0.0%	0.50e00 0.0%

Scopes	CPU_Time	GainExtraRes	GainExtraILP	GainVectorize	CPUBottleNeck	CPU_U_Load[0]	CPU_U_StAddr[0]
Experiment Aggregate Metrics	4.76e09 100.0	2.68e09 100.0	1.72e09 100.0	2.34e09 100.0	2.15e09 100.0	1.72e09 100.0	4.29e08 100.0
main	4.76e09 100.0	2.68e09 100.0	1.72e09 100.0	2.34e09 100.0	2.15e09 100.0	1.72e09 100.0	4.29e08 100.0
loop at source.c: 14–18	4.76e09 100.0	2.68e09 100.0	1.72e09 100.0	2.34e09 100.0	2.15e09 100.0	1.72e09 100.0	4.29e08 100.0
loop at source.c: 14–17	4.76e09 100.0	2.68e09 100.0	1.72e09 100.0	2.34e09 100.0	2.15e09 100.0	1.72e09 100.0	4.29e08 100.0
loop at source.c: 16–18	4.76e09 100.0	2.68e09 100.0	1.72e09 100.0	2.34e09 99.9%	2.15e09 100.0	1.72e09 100.0	4.29e08 100.0
loop at source.c: 16–18	2.58e09 54.1%	2.08e09 77.3%	5.73e08 33.3%	5.73e08 24.4%	1.72e09 80.0%	1.72e09 100.0	
loop at source.c: 16–18	8.23e08 17.3%	3.58e08 13.3%	3.94e08 22.9%	7.16e08 30.6%	3.58e08 16.7%		3.58e08 83.3%
loop at source.c: 16–18	5.37e08 11.3%	7.16e07 2.7%	3.94e08 22.9%	5.01e08 21.4%	7.16e07 3.3%		7.16e07 16.7%

Fig. 8. MIAMI results for the matrix multiply code compiled with gcc (top), and icc (bottom)

B. *PLASMA Kernel*

For our second experiment, we look at the math kernel `CORE_dtsmqr` from the optimized linear algebra library for multicore architectures, *PLASMA* [5]. One of the *PLASMA* developers remarked during one private conversation that some of the core kernels in *PLASMA* do not run as efficiently (as high a fraction of machine peak FLOPS) as the well known `dgemm` kernel, especially at smaller problem sizes. Small problem sizes, such as `128x64` per core, are desired for the increase in thread and task level parallelism that they provide.

We offered to analyze one of these kernels, and we were pointed in the direction of the `CORE_dtsmqr` kernel, which is used by multiple solvers. *PLASMA* can be configured to use external *BLAS* libraries. On Intel multicores, *PLASMA* is typically configured to use the optimized Intel MKL libraries. We used MIAMI to analyze the performance of `CORE_dtsmqr` with input parameters `nb` and `ib` set to `128` and `64`, respectively, which was the configuration requested by the developers. Figure 9 presents a snapshot of some of the metrics computed by MIAMI for the top time consuming routines. All these routines are inside the Intel MKL library.

We notice that the top two routines, `mkl_blas_avx_dgemm_kernel_0` and `mkl_blas_avx_sgem2vu_even` account for 68% and 18% of the instruction execution cost, respectively, based on our model. The seven metrics included in the figure are, in order: execution time, improvement potential from additional machine resources, improvement potential from additional ILP, improvement potential from vectorization, total time due to resource contention, time due to contention on the `U_FpAdd` unit, and time due to contention on the `U_FpMult` unit. The performance of the top two routines is limited primarily by contention on machine resources. The `dgemm` kernel's performance is limited by the availability of the floating-point adder unit, and the `sgem2vu` kernel's performance is limited by the availability of the multiply unit. The execution unit usage report produced by MIAMI shows that both kernels make balanced use of the adder and the multiply units. The first kernel executes just a few extra additions, and the second kernel executes just a few extra multiply instructions.

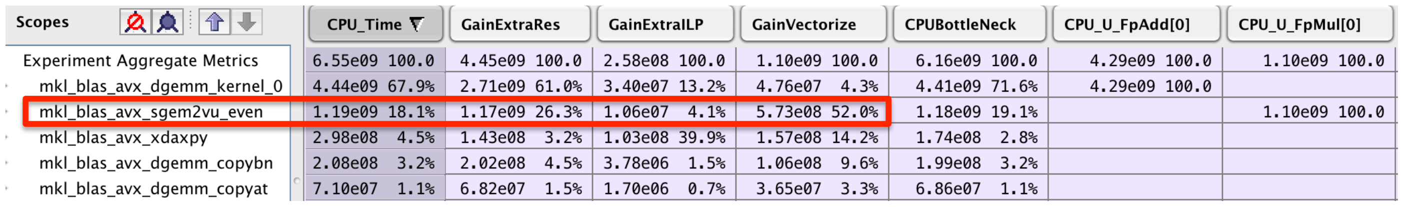
As expected, both kernels would benefit very little from additional instruction level parallelism. However, when we look at the improvement potential from vectorization, we see a different picture. The `dgemm` kernel's improvement potential from vectorization is barely larger than its improvement potential from higher ILP, suggesting that the kernel is well vectorized. This observation was confirmed by inspecting the instruction mix report. On the other hand, the `sgem2vu` kernel's improvement potential from vectorization is much higher than its improvement potential from additional ILP, suggesting that the kernel is not well vectorized. When we looked at this kernel's instruction mix report, we observed that it included only 128-bit vector instructions. Thus, while the kernel has a balanced mix of instructions that results in a high utilization of the floating-point adder and multiply units, it can run at no more than 50% of peak.

While our findings do not provide a direct path for the *PLASMA* developers to fix this performance inefficiency since the MKL library is closed source, it provides a clear answer to the question of why their kernel is not running at peak FLOPS.

C. *LULESH 2.0.2*

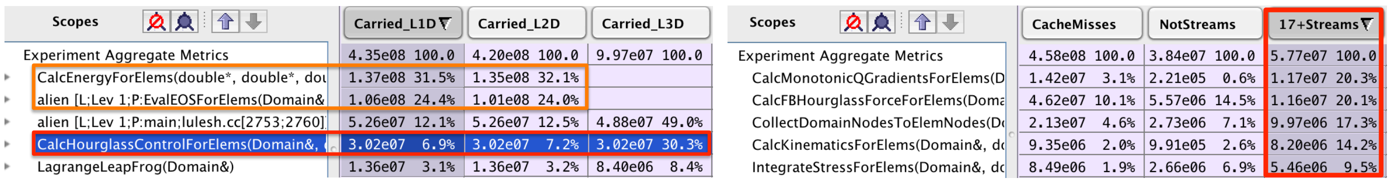
For our final case study, we analyze a serial run of the hydrodynamics proxy application LULESH 2.0.2. LULESH operates on an unstructured hexahedral mesh and is more memory intensive than our first two case studies. We focus our narrative on the insight provided by MIAMI about data reuse patterns and prefetcher unfriendly memory accesses.

Figure 10 shows memory analysis results for a serial run of LULESH using a mesh of size 45 and 50 time steps of simulation. We used the MIAMI data locality analyzer to understand the memory reuse patterns in LULESH. Figure 10a displays the program scopes that *carry* the most misses in each of the three cache levels on our target system. A scope S *carries* cache misses generated by data reuse patterns for which S is the carrying scope (see Section III-D). Cold misses are not included. This means that the program control flow returns back to scope S between two consecutive accesses to a particular data D, and the second access to D has a memory reuse distance that is too long for our target cache. Identifying



Scopes	CPU_Time	GainExtraRes	GainExtraILP	GainVectorize	CPUBottleNeck	CPU_U_FpAdd[0]	CPU_U_FpMul[0]
Experiment Aggregate Metrics	6.55e09 100.0	4.45e09 100.0	2.58e08 100.0	1.10e09 100.0	6.16e09 100.0	4.29e09 100.0	1.10e09 100.0
mkl_blas_avx_dgemm_kernel_0	4.44e09 67.9%	2.71e09 61.0%	3.40e07 13.2%	4.76e07 4.3%	4.41e09 71.6%	4.29e09 100.0	
mkl_blas_avx_sgem2vu_even	1.19e09 18.1%	1.17e09 26.3%	1.06e07 4.1%	5.73e08 52.0%	1.18e09 19.1%		1.10e09 100.0
mkl_blas_avx_xdaxpy	2.98e08 4.5%	1.43e08 3.2%	1.03e08 39.9%	1.57e08 14.2%	1.74e08 2.8%		
mkl_blas_avx_dgemm_copybn	2.08e08 3.2%	2.02e08 4.5%	3.78e06 1.5%	1.06e08 9.6%	1.99e08 3.2%		
mkl_blas_avx_dgemm_copyat	7.10e07 1.1%	6.82e07 1.5%	1.70e06 0.7%	3.65e07 3.3%	6.86e07 1.1%		

Fig. 9. MIAMI results for kernel `CORE_dtsmqr` from the *PLASMA* library



Scopes	Carried_L1D	Carried_L2D	Carried_L3D
Experiment Aggregate Metrics	4.35e08 100.0	4.20e08 100.0	9.97e07 100.0
CalcEnergyForElems(double*, double*, dou	1.37e08 31.5%	1.35e08 32.1%	
alien [L;Lev 1;P:EvalEOSForElems(Domain&	1.06e08 24.4%	1.01e08 24.0%	
alien [L;Lev 1;P:main;lulesh.cc[2753;2760]	5.26e07 12.1%	5.26e07 12.5%	4.88e07 49.0%
CalcHourglassControlForElems(Domain&,	3.02e07 6.9%	3.02e07 7.2%	3.02e07 30.3%
LagrangeLeapFrog(Domain&)	1.36e07 3.1%	1.36e07 3.2%	8.40e06 8.4%

(a) Memory reuse insight

Scopes	CacheMisses	NotStreams	17+Streams
Experiment Aggregate Metrics	4.58e08 100.0	3.84e07 100.0	5.77e07 100.0
CalcMonotonicQGradientsForElems(D	1.42e07 3.1%	2.21e05 0.6%	1.17e07 20.3%
CalcFBHourglassForceForElems(Doma	4.62e07 10.1%	5.57e06 14.5%	1.16e07 20.1%
CollectDomainNodesToElemNodes(Do	2.13e07 4.6%	2.73e06 7.1%	9.97e06 17.3%
CalcKinematicsForElems(Domain&, do	9.35e06 2.0%	9.91e05 2.6%	8.20e06 14.2%
IntegrateStressForElems(Domain&, do	8.49e06 1.9%	2.66e06 6.9%	5.46e06 9.5%

(b) Data streaming insight

Fig. 10. Memory analysis results for a serial run of LULESH 2.0.2

the carrying scope of a reuse pattern is very important for tuning. To improve the locality of a data reuse pattern, we have to apply code transformations starting from the carrying scope, as explained in Section III-D. MIAMI captures detailed information about each data reuse pattern, including source, destination and carrying scopes. We do not include screenshots of all this data due to lack of space, but we explain the relevant details in text.

Misses in the last level of cache are generally the most costly for an application. We start our analysis by looking at metric `Carried_L3D`. We notice that 49% of L3 cache misses are carried by a level 1 loop in the `main` routine. This loop is the main time step loop of the simulation. These misses are generated by reuse of data across iterations of the main time step loop, and are difficult or impossible to eliminate due to likely data dependencies. Routine `LagrangeLeapFrog` carries 8.4% of L3 cache misses. This routine, called from the main time step loop, coordinates the simulation of one time step. Misses carried by this routine are between the different phases of a simulation step and are also difficult to eliminate.

Routine `CalcHourglassControlForElems` carries 30% of L3 cache misses between a level 1 loop in the same routine (L_1) and a level 1 loop of a direct callee (L_2). Both loops iterate over all the mesh elements. L_1 computes intermediate values for each of the eight nodes associated with a mesh cell and stores them in six temporary arrays. L_2 uses these intermediate values to compute nodal forces. We can improve such data reuse patterns by fusing the two loops, which would eliminate the need for temporary arrays. Because the two loops are located in different routines, we improved their locality by tiling both loops with the same tile size, moving the loops over tiles to the `CalcHourglassControlForElems` routine and fusing them. The new code uses the same six temporary arrays, however, the array sizes are proportional to the tile size. By shortening the temporary arrays, we reduced the application's memory footprint and removed an equal number of cache misses carried across iterations of the main time step loop between L_2 and L_1.

Metrics `Carried_L1D` and `Carried_L2D` in Fig. 10a show that routine `CalcEnergyForElems` carries 32% of L1 and L2 cache misses, while a level 1 loop in routine `Eval-EOSForElems` carries an extra 24% of L1 and L2 misses. Routine `CalcEnergyForElems` carries misses between different loop nests located in the same routine. The level 1 loop in routine `EvalEOSForElems` implements an artificial load imbalance in LULESH by repeating the calculations in this routine a variable number of times, depending on the region index. This loop includes several inner loops of its own and a call to routine `CalcEnergyForElems`. It carries misses both within- and across-iterations, between its inner loops and loops located in its callee. The code uses a large number of temporary arrays of size equal to the number of elements in a region. Such data reuse patterns can be improved by fusing the various inner loops. Instead of fusing the loops, we found it easier to preserve the code structure. As before, we tiled the inner loops in these two routines, promoted the loops over tiles to the caller routine and fused them. We kept the temporary arrays, however, their sizes are much shorter.

Figure 10b shows some of the insight provided by MIAMI about prefetcher unfriendly memory accesses in LULESH. The figure shows the number of L1 misses, the number of misses that are not part of streaming access patterns, and the number of misses part of high concurrency streams. Non-streaming memory accesses are generally caused by irregular memory access patterns, and they cannot be effectively prefetched by streaming hardware prefetchers. Regions with high streaming concurrency correspond to program loops that have many independent streams. Hardware prefetchers can track only a limited number of data streams due to finite hardware resources. For example, the AMD 10H hardware prefetchers are effective for up to only 16 concurrent streams [17]. We notice that roughly 12.6% of all L1 cache misses ($5.77e07$ out of $4.58e08$ misses) correspond to high concurrency streaming accesses. The top five routines contributing to this metric account for over 80% of high concurrency accesses. These routines access various arrays associated with mesh nodes that store node coordinates, velocities, accelerations and nodal forces. Each of these kinematic values have x, y and z contributions stored in separate arrays. We replaced the three arrays associated with each kinematic nodal metric, as well as the three arrays storing element strains, with arrays of structures with three fields.

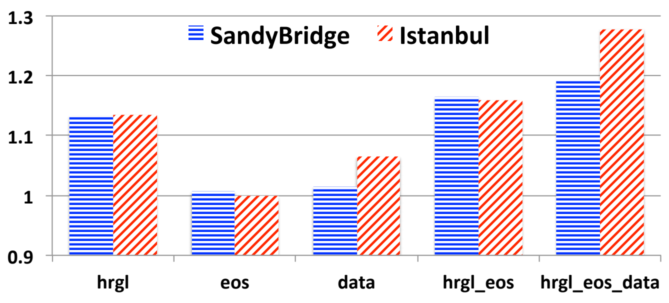


Fig. 11. LULESH speedup on Intel SandyBridge and AMD Istanbul systems

Figure 11 presents a summary of the speedups obtained after applying the code transformations described above on our target SandyBridge system and on an AMD Istanbul machine at 2.6 GHz. Each column cluster shows data for one code variant. Thus, `hrgl` represents the tiling & fusion transformation applied to routine `CalcHourglassControlForElems`, `eos` represents the tiling & fusion transformation applied to routine `EvalEOSForElems`, and `data` represents the fusion of the data arrays. We also show speedups achieved by the combination of the two data reuse optimizations, and by the combination of all three transformations. The two data reuse optimizations yield very similar speedups on both systems. As expected, version `hrgl` yields the highest speedup of all three transformations. Version `eos` achieves no speedup by itself, but it yields a modest speedup when combined with the `hrgl` transformation. The fusion of data arrays yields a significantly higher speedup on the AMD system and accounts for the difference in total speedup observed on the two systems.

The feedback provided by MIAMI pinpointed several low-hanging tuning opportunities, even as we had no prior experience with the application. In a typical tuning exercise, one would repeat the analysis on the newly optimized code to identify the next largest opportunities for improvement.

V. Related Work

Many performance analysis tools and performance modeling techniques have been proposed by the research community over the years to tackle performance analysis. These tools fall broadly into two categories: 1) performance measurement tools that use either instrumentation [3], [21] or hardware counter sampling [1], [10], [20] to measure performance effects on a particular architecture; and 2) performance modeling and performance prediction tools [19], [22], [25].

Hardware counter measurements are useful for getting an accurate view of performance effects occurring on a real architecture. Tools such as [1], [3], [10], [20], [21] facilitate the collection of hardware counter measurements and are useful for identifying hotspots in applications. Some of these tools also compute a call graph of an application, which helps users identify the callers of a hotspot routine. However, performing root cause analysis from hardware counter measurements requires an additional process of deconvolution and good knowledge of the underlying architecture. The diagnostic information produced by MIAMI complements hardware counter measurements and provides additional insight, currently unavailable through raw measurements.

PerfExpert [4] uses HPCToolkit to collect time and cache miss profiles of an application. Next, it analyzes the profile data to find hotspots, places where most time is spent or where most cache misses are incurred. It outputs a list of the identified hotspots in text format, and a list of typical loop nest transformations that a user can try. The code transformation suggestions are generic, not tailored to the particular memory access patterns in the application, which limits their usability.

Existing performance modeling approaches either use application operation counts and machine peak performance rates to compute upper bounds on achievable performance, or use agnostic models that can predict a metric of interest while being oblivious of system internals, and thus cannot provide diagnostic insight. The Roofline model [25] uses an application's arithmetic intensity, the ratio between the number of executed floating-point operations and the number of bytes transferred from memory, to classify the application as compute bound or memory bound. The model can compare achieved application performance against an intuitive, visual model of machine peak performance rates. However, if an application does not already perform close to machine peak, the model does not provide any real insight into the factors that limit its performance or guidance on how to change its arithmetic intensity.

PBound [19] relies on static analysis of an application's source code to collect operation statistics. The source code metrics are then convolved with an abstract architectural model to compute performance-related estimates, e.g., execution time or cache miss rates. Because the analysis is limited to source code, the bounds are not tight in the presence of irregular memory access patterns and complex dynamic behavior.

Prophesy [22] uses empirical measurements of application performance for different inputs, and applies curve fitting to compute a scalable model of performance as a function of application input parameters. The models are application and architecture specific. This is an example of an agnostic model that can predict performance for a different program input while not modeling any of the system's internals. Such models provide little performance diagnostic insight.

Beyls and D'Hollander [2] describe RDVIS, a tool for visualizing reuse distance information clustered based on the intermediary executed code (IEC) between two accesses to the same data, and SLO, a tool that suggests locality optimizations based on the analysis of the IEC. The capabilities of their tools are similar to our data locality analyzer. However, our implementations differ in the ways we collect and analyze the data. RDVIS uses compiler based instrumentation, while MIAMI works on x86 executables, which enables it to observe effects of compiler optimizations and interactions between application code and third party libraries. Fauzia et al. [13] construct a computational directed acyclic graph (CDAG) of data dependences in an applications. They partition the CDAG into convex subsets, and use the partitions to find instruction orderings that improve locality while preserving data dependencies. Their analysis confirms the existence of better, valid instruction orderings, but does not provide direct guidance on how to transform the code to obtain those orderings. In contrast, the MIAMI data reuse analyzer provides insight into the type of code transformations needed to improve a data reuse pattern, however, our analysis does not determine automatically if all transformations are legal.

VI. Conclusions

Performance diagnosis has been a largely manual process. While many tools support collection of performance profiles and traces, analysis of this data to understand sources of inefficiency is typically left for the user. The difficulty of this task is compounded by the fact that current tools focus on collecting performance effects, the result of interactions between code and a particular architecture. A process of deconvolution through which we can attribute parts of the observed effects to specific application and architectural factors is needed to perform root cause analysis from such measurements.

In this paper, we introduce MIAMI, a collection of tools for understanding performance inefficiencies in applications. Instead of measuring performance effects, MIAMI collects profiles of application characteristics that are largely architecture independent. MIAMI uses detailed static analysis and performance modeling based on first order principles to reason about performance and to pinpoint sources of inefficiency in applications. MIAMI computes a large number of performance metrics, focused primarily on facilitating performance diagnosis. One of MIAMI's main goals is to eliminate or drastically reduce the amount of guesswork traditionally involved in uncovering tuning opportunities. The three case studies presented in the paper validate many of our design decisions. The matrix multiply and the PLASMA kernel examples underscore the importance of working at the executable level to capture the effects of compiler optimizations and to enable analysis of third party libraries for which source code may not be available. The LULESH example demonstrates how the insight provided by MIAMI can pinpoint low-hanging tuning opportunities, even when one has no prior experience with an application.

Currently, MIAMI outputs a set of detailed performance metrics. An experienced user that can interpret this data would get the most benefits from the provided insight. Our plans for the future call for automating the interpretation of performance results to make the tools accessible to a wider range of users, and on extending our diagnosis approach to accelerators such as the Intel Phi and GPUs.

VII. Acknowledgments

We thank the anonymous reviewers for their improvement suggestions. This work was funded, in part, by the Office of Advanced Scientific Computing Research in the U.S. Department of Energy under Grant No. DE-SC0006733, by the National Science Foundation under Grant No. CNS-0910899, and by the Department of Defense under Grant No. XXX-YYYY-XXXXXX.

References

[1] 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 http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, Apr. 2010. 1, 6, 10

[2] K. Beyls and E. H. D'Hollander. Intermediately executed code is the key to find refactorings that improve temporal data locality. In *CF '06: Proceedings of the 3rd Conference on Computing Frontiers*, pages 373–382, New York, NY, USA, 2006. ACM Press. 5, 10

[3] S. Browne, C. Deane, G. Ho, and P. Mucci. PAPI: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, 1999. 1, 10

[4] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. 10

[5] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1):38–53, Jan. 2009. 6, 8

[6] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, 1988. 6

[7] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, Aug. 1988. 5

[8] E. Chung, P. Milder, J. Hoe, and K. Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 225–236, 2010. 2

[9] I. Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html [27 September 2013]. 3, 4

[10] I. Corporation. Intel VTune Amplifier XE 2013. http://software.intel.com/en-us/intel-vtune-amplifier-xe [04 October 2013]. 1, 10

[11] I. Corporation. XED. http://software.intel.com/sites/landingpage/pintool/docs/61206/Xed/html/ [27 September 2013]. 3

[12] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM. 2

[13] N. Fauzia, V. Elango, M. Ravishankar, J. Ramanujam, F. Rastello, A. Rountev, L.-N. Pouchet, and P. Sadayappan. Beyond reuse distance analysis: Dynamic analysis for characterization of data locality potential. *ACM Trans. Archit. Code Optim.*, 10(4):53:1–53:29, Dec. 2013. 10

[14] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, Aug. 1992. 5

[15] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, LLNL, August 2013. 6

[16] 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 *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. 3

[17] G. Marin, C. McCurdy, and J. S. Vetter. Diagnosis and optimization of application prefetching performance. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 303–312, New York, NY, 2013. 6, 9

[18] G. Marin and J. Mellor-Crummey. Pinpointing and exploiting opportunities for enhancing data reuse. In *Proceedings of the 2008 IEEE International Symposium on Performance Analysis of Systems and Software*, Apr 2008. 5

[19] S. H. K. Narayanan, B. Norris, and P. D. Hovland. Generating performance bounds from source code. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 197–206, Washington, DC, USA, 2010. IEEE Computer Society. 10

[20] The OProfile website. http://oprofile.sourceforge.net/docs. 1, 10

[21] S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications, SAGE Publications*, 20(2):287–331, 2006. 1, 10

[22] V. Taylor, X. Wu, and R. Stevens. Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. *SIGMETRICS Perform. Eval. Rev.*, 30(4):13–18, Mar. 2003. 10

[23] TOP500 Supercomputer Sites. http://www.top500.org. 1

[24] J. Vetter, S. Lee, D. Li, G. Marin, C. McCurdy, J. Meredith, P. Roth, and K. Spafford. Quantifying architectural requirements of contemporary extreme-scale scientific applications. In *4th International Workshop on Performance Modeling, Benchmarking and Simulation of HPC Systems (PMBS13)*, Denver, Colorado, November 2013. 2

[25] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009. 10

[26] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991. 5