Beyond Profiling

Chris Quackenbush
Google
cquackenbush@gmail.com

Mohamed Zahran
Computer Science Dept, NYU
mzahran@cs.nyu.edu

Abstract

Profiling techniques are used extensively at different parts of the computing stack to achieve many goals. One major goal is to make a piece of software execute more efficiently on a specific hardware platform, where efficiency spans criteria such as power, performance, resource requirements, etc. Researchers have introduced many techniques to gather, and make use of, profiling data. However, one thing remains unchanged: making application "A" run more efficiently on machine 1. In this paper, we extend this criterion by asking: can profiling information of application "A" on machine 1 be used to make application "B" run more efficiently? If so, then this means that as machine 1 continues to execute more applications, it becomes better and more efficient.

We present a genlasteralized method for using profiling information gathered from the execution of programs from a limited corpus of applications to improve the performance of software from outside our corpus. As a proof of concept, we apply our technique to the specific problem of selecting the most efficient last-level-cache size with which to execute an application. We were able to turn off an average of 38% of last-level-cache blocks from PARSEC benchmark suite and only saw an average 0.30% increase in the rate of last-level cache misses leading to negligible execution time overhead.

1. Introduction

There are many techniques for gathering and utilizing profiling data, but they all share a common constraint: each software application must be instrumented and executed on a hardware platform at least once to gather profiling information before any useful optimizations can be made. We propose that **there is a finite set of patterns along which hardware/software interactions can occur to give best performance**. For example, given a cache configuration, there are finite set of memory access patterns that yield low cache misses. Or, given a memory access pattern, we can build the best cache configuration that yields the lowest number of misses. We can determine the best hardware configuration for each interaction pattern by capturing profiling data from a corpus of representative applications on a small set of hardware configurations. We can then build a system which recognizes these patterns while executing applications from outside of the corpus and uses the most performant hardware configuration. The more profiling data is available, the more patterns will be recognized and the better the execution.

This method can be compared with other techniques for optimizing performance during application execution. Optimizing compilers are one of the most common forms of performance enhancement, but these techniques are forced to be very general because the actual program behavior at runtime is unknown [10]. Profile-guided compilation [19] uses data from previous executions to select more appropriate optimizations, but requires the application to be run and profiled at least once before any performance benefits can be realized [19]. More broadly, all techniques that require the program to be recompiled add additional overhead before running the application.

Other feedback-directed optimizations such as branch prediction [22] can be done at runtime, but suffer from the weakness that these optimizations must be based on the behavior of the application in its earlier phase of execution. In many programs, the same subroutine or basic block may have different behavior over the course of execution [19] depending on the data being processed.

Using the method presented in this paper, profiling data from previous applications can be reused without ever profiling the performance of a target application, but only observing its interaction pattern with the hardware. This allows us to provide profile-guided optimizations to all applications the first time they are executed. Moreover, as we collect more profiling information, our presented method can become more and more precise.

Using the profile data to make an execution of a program more efficient requires two things: a learning mechanism (we will discuss that in Section 3) and reconfiguring either the program binary or the hardware. We chose the latter and discuss it in Sections 4 and 5.

2. Related Work

Previous work on improving performance through profiling has been divided into two distinct categories: those techniques that modify the hardware platform and those that modify the application software or software platform. Chen et al [14] used supervised learning to predict detailed software profiles from coarse granularity hardware event sampling data (e.g. L2 cache write misses). Han and Abdelrahman [6] used a machine learning model trained on benchmarks to successfully predict when local memory usage in GPUs would be an effective optimization. Ipek et al [7] used a similar technique to choose hardware configurations in an offline way. They presented a method for using sampled hardware performance and an artificial neural network to predict performance of unseen hardware designs so as to efficiently explore of the architectural design space when creating multiprocessors.

Other methods for reconfiguring hardware online as a program executes which make use of feedback-directed optimization techniques, but do not use machine learning have also appeared. Abella et al [8] presented a heuristic for predicting the last instruction accessing each L2 cache block and disabling the blocks to reduce power consumption. Banerjee, Subhasis, and Nandy [9] turned off entire cache ways based on the observed miss rate as an application executes.

Using profiling information from previous runs of the *same* program to produce better future executions without recompiling the application is not a new technique. Schulte et al [13] presented a method for using genetic algorithms to continually profile and transform application object code to increase power efficiency across multiple executions. Work has also been done to use profiling information from applications to reconfigure the execution environment. Yuan, Guo, and Chen [12] used profile-guided optimization to recompile the Linux kernel (as opposed to the application software) to produce a more performant environment for user software.

3. The Proposed Idea

We present a method for using profiling information gathered from the execution of programs from a limited corpus of applications to improve the performance of software from outside our corpus. That is, we answer this question: how can profiling information from a small set of known programs be used to improve the performance of all unknown programs? Our proposed technique is performed in two phases: a training phase that occurs once, and offline, and produces a model of all profiling data from the training corpus; and an execution phase that occurs while the target, unseen, application is running and reconfigures the system based on the observed pattern to the optimal state for executing the current program.

During the training phase, profiling information is collected as each corpus application executes in each system configuration. The number of configurations is can be small or large depending what you try to configure. In this paper, as a proof-of-concept, we choose to configure the size of LLC by turning off some cache slots. So, we picked 5 configurations (100% of LLC is on, 80%, 60%, 40%, and 20%). During each execution, signatures summarizing the hardware/software interaction pattern are collected (more on signatures in Section 4). These interactions could include cache behavior, instruction pipeline behavior, memory behavior, etc, depending on the hardware criteria we want to tackle. The signatures are combined with the profiling data to build a model which maps each observed execution signature to an optimal system configuration (as determined by the profiling data). The job of the model is to map the optimal configuration from an observed signature. A model could be a simple as a hash map of signatures or as complicated as a neural network.

During the execution phase, signatures are collected continually as the hardware/software interaction patterns occur. These signatures are consumed by a configuration selection algorithm which uses the model of profiling data created in the training phase to predict which system configuration will be most performant. When a new configuration is predicted to be more performant than the current configuration, the system is immediately reconfigured to the state selected by the algorithm. This process continues until the target application ends.

4. Reconfiguring the Hardware Platform

To illustrate our method, we provide a system to perform the following task: Given a running candidate application, that was not profiled before, and predefined set of hardware configurations, select the best hardware configuration with which to execute the application. This is repeated at each phase [1] during application execution. We first define a cost function which will specify the hardware configuration that performs best. By selecting the configuration that minimizes this function we will make the hardware platform more performant along the dimension of the cost function. The cost function can measure execution time, power consumption, resource utilization, or any other chosen metric. The training phase does not need to be exhaustive. As new profiling data are available, another round of training can be done. As we will show in the results, the more profiling data, the better the future executions of different programs. Therefore, making a better use of all profiling data that are collected in all systems.

During the training phase, each application from the corpus will be executed on each predefined hardware configuration*. The resulting profiling data will be analyzed with respect to the cost function to determine which hardware configuration is the most efficient and a model will be built mapping the observed signatures to that hardware state. As the target application executes, that model is used to predict the optimal hardware configuration and the hardware platform is continually reconfigured, at each phase we will be shown, to the state determined to be most efficient for the current application behavior.

4.1 Program Phases and Working Sets

Application software typically exhibits long sequences of uniform behavior interrupted by abrupt periods of unstable activity [15]. These periods of stability are called program phases [23]. We can divide each phase into periods of equal length (as measured by any regularly occurring event, such as CPU cycles, instructions committed, memory accesses, etc). We will call each of these periods an execution window. The working set for each execution window is defined as the set of distinct resources (typically memory regions) accessed during that window [20].

Program phases are primarily caused by working sets and the abrupt phase change is a reflection of a change in the working set [1]. The working set of instructions is defined by the control passing through subroutines and nested loops. The set of instructions executed is very similar in some intervals and suddenly different as control exits from a loop or subroutine [11].

4.2 Working Set Signatures

A working set signature (WSS) is a compressed representation of the entire working set over a single window. An instructions WSS is related to cache performance measurements (e.g. miss rate), but because it is independent of the hardware interaction it doesn't capture details of the specific memory implementation or geometry [5]. A memory WSS relates directly to the blocks or pages

* The method presented in this paper can be used with any hardware aspect (we used LLC size as an example), such as: branch prediction table size, branch prediction algorithm, scheduling of instructions for execution, cache replacement policy, etc. The design space is manageable for each of these aspects, and since it is done offline then there is no performance loss.

accessed during the window and can therefore capture more details of the software/hardware interaction.

$$\frac{hamming\_weight(WSS1 \oplus WSS2)}{hamming\_weight(WSS1 \mid WSS2)} \quad (1)$$

The distance function between two WSS is shown in Equation (1). The WSS distance function counts the number of one bits in the bitwise exclusive-or of two signatures and the one bits of the bitwise inclusive-or and expresses the ratio of the two counts. The *hamming_weight* function is the sum of all the bits in its argument (i.e. the number of one bits).

4.3 Memory Access Signatures

A memory access signature (MAS) is a record of the recent performance for a unit of cache memory. We use a MAS of 64 bits wide (typical size of registers in current machines, without loss of generality) and shifted left each time the cache unit is accessed. When the access results in a cache hit, the least significant bit is set to one; on a cache miss, the least significant bit is set to zero.

5. Reconfiguring the Last-Level Cache

5.1 The Simulated Hardware Platform

In the following sections, we use the motivating example of selecting a last-level-cache size from a small set of available cache sizes to minimize the amount of cache turned on during execution hence saving power. This approach can then be generalized to tuning other hardware parameters.

We use the Multi2Sim system simulation framework [17] to simulate the execution of our corpus and candidate applications. The machine we simulated uses a single X86 CPU with a single thread and a memory of two-level caches. L1 (one for data data and one for instructions) is 2-way set associative, 2 cycles latency, and total size of 32KB. L2 is 16-way, 10 cycles delay, and total size of 1MB. Block size is fixed at 64 bytes. We use this simple configuration to isolate the effect of profile data of a program on the performance (of the chosen criteria) of another program.

5.2 Reconfigurable Last Level Cache

We have extended Multi2Sim to support the ability to turn off individual cache blocks in the last level cache (in the case of these experiments the LLC is always the L2 cache). Our system can reduce the number of accessible cache blocks in the LLC to any number greater than the number of sets. To ensure that the entire main memory remains cacheable, this system will never disable all blocks in a set. This constraint is the reason that our system does not allow for the number of blocks in the LLC to be reduced below the number of sets. Otherwise, we will need to change the number of sets, which will make the cache controller more complicated, and hence slower and power hungry.

ALGORITHM 1: LAST LEVEL CACHE RECONFIGURATION

```
FUNCTION (blocks_to_turn_off)
  pass = 0
  WHILE (blocks_to_turn_off > 0)
    pass += 1
    FOREACH way in LLC_ASSOCIATIVITY
      FOREACH set in LLC_ROWS
        block = set[way]
        IF exactly one way of set is turned on
          continue
        IF block is accessed AND pass < 2
          continue
        IF block is dirty AND pass < 3
          continue
        turn off block
        Blocks_to_turn_off -= 1
```

Algorithm 1 describes the procedure for choosing blocks to turn off. The input to the algorithm, blocks_to_tun_off, is decided by the configuration algorithm. This procedure minimizes the cost of decreasing the cache size by first preferring to disable blocks that have never been accessed and then disabling blocks that have never been written; only once all other blocks have been disabled will it begin to incur the overhead of turning off dirty cache blocks.

To implement Algorithm 1 in hardware we use two bits per cache block: accessed bits and dirty bit. These two bits are already available in most LLC to implement the replacement policy. Each cache set also maintains a count of the number of blocks that are turned on in that set and no set is ever allowed to turn off all its blocks. The cache blocks are turned off in three passes and in each pass several blocks are turned off simultaneously. The first pass turns off blocks that neither accessed nor dirty. If a second pass is needed, blocks that are not dirty table are turned off. The third pass turns off any remaining blocks. The LLC will never disable more than the required number of blocks and blocks with a lower way index are preferred to spread the disabled blocks across all the LLC sets.

Although this algorithm required three rounds, only the third round needs to block pipeline execution (as the dirty blocks are written back to memory). The first two rounds happen in parallel with application execution. In practice, we found that the third round is very rarely needed; we never observed it in any of our experiments.

5.3 Cache Size Selection Algorithms

We have implemented three algorithms in Multi2Sim for selecting the the number of blocks to turn off in the last-level cache: the extended working set signatures algorithm [2], one that uses a bloom filter [21] as the profiling model (BLOOM), and one that uses an artificial neural network [18] as the profiling model (ANN). BLOOM and ANN are new procedures which we implemented to utilize profiling data collected from our application corpus while EWSS makes no use of profiling data. We will use EWSS as a baseline procedure against which to compare our new methodology in addition to the base configuration where each block in the LLC is always turned on.

5.4 The Baseline Algorithm

As a baseline against which to compare our new methodology, we implemented the extended working set

signatures method (EWSS) introduced by Dhodapkar and Smith [2] for selecting a cache size. This algorithm iterates over all available hardware configurations and uses online hardware event sampling in each configuration to determine the optimal hardware state. The EWSS algorithm uses the instruction cache working set and is based on earlier work by Balasubramonian et al [3] which was presented for adjusting the L1 cache size. A very similar algorithm was designed by Zhang, Vahid, and Lysecky [4] for tuning cache memory along multiple parameters in embedded systems via an exhaustive search of the hardware configuration space during execution.

EWSS does not make use of any corpus of profiling data and does not have a training phase to build a model for predicting the optimal configuration. Instead, each time a new program phase is entered, EWSS iterates over all available configurations and profiles application behavior in each one; it then reconfigures the system to the state observed to be more performant. As the basis for its program phase detection, EWSS uses a working set signature derived by hashing the address of each instruction fetched from the instruction cache.

ALGORITHM 2: EXTENDED WORKING SET SIGNATURE (EWSS)

```
prev_wss, wss: 128 bytes initialized to zero

During each window of 100,000 instructions:
index = hash of instruction_pointer into the
range [0, 1024]
wss[index] = 1

After each 100,000 instruction window:
distance = WSS_DISTANCE(prev_wss, wss)
IF distance > 0.5
  unstable_windows++
  IF unstable_windows > 10
    stable_windows = 0
    set LLC to maximum size
ELSE
  stable_windows++
  IF stable_windows > 4
    unstable_windows = 0
    IF all LLC configurations were tested in the
current phase
      set LLC to best configuration
    ELSE
      set LLC to next configuration
prev_wss = wss
wss = 0
```

The EWSS method uses instruction cache WSS distance (Equation 1) to detect stability across program execution windows and reconfigures the LLC each time it detects a new program phase is entered. Once a new phase is detected via stability in the WSS distance the EWSS algorithm iterates over every available size of the LLC (one per window) and selects the most efficient size for the remainder of the program phase. In our case, efficiency is determined by the size with the fewest cache misses.

5.5 The Bloom Filter Algorithm

The first procedure we implemented that takes advantage of profiling data from our corpus is described in Algorithm 3. This method populates a bloom filter [21] for each hardware configuration with every observed memory access signatures (MAS) that performed best with that configuration.

As described in Section 6, the corpus of training applications is used to generate a set of MAS and profiling data is used to associate each MAS with an optimal hardware configuration. The model used in the BLOOM algorithm is a set of bloom filters, one corresponding to each hardware configuration. Each observed MAS is added to the bloom filter corresponding to its associated optimal hardware configuration.

Table 1: The applications of the training corpus used to collect profiling data

Application	Description	Data Sizes
list	Builds a doubly linked list and then traverses it from head to tail	9, 100, and 1024 nodes
sort	Creates an array of randomly generated 32-bit integers and performs quicksort on the data	9, 100, or 1024 integers
transp	Creates an array of randomly generated 32-bit integers and, treating it as a square matrix, populates an identically sized array with the transpose of that matrix.	9, 100, or 1024 integers interpreted as a 3x3, 10x10, or 32x32 square matrix
mul	Creates two arrays of randomly generated 32-bit integers and, treating them as a square matrices, populates an identically sized array with the product of those matrices.	18, 200, or 2048 integers interpreted as a two 3x3, 10x10, or 32x32 square matrices

ALGORITHM 3: BLOOM FILTER MEMORY ACCESS SIGNATURES (BLOOM)

```
bloom_filter_list: array of one filter per LLC
configuration

Populate bloom filters during offline training:
FOREACH mas in all MAS observed in training
corpus
  best_configuration = best observed
configuration for MAS
  add_to_filter(mas,
bloom_filter_list[best_configuration])

Each time the LLC is accessed:
mas = updated MAS
FOREACH configuration in LLC configurations
  IF check_filter(mas,
bloom_filter_list[configuration]
    set LLC to configuration
    return
```

In our experiment, the cost function was selected to minimize the LLC size without increasing the number of execution cycles.The implementation of this function will be described in section 6.3.

During execution of the candidate software, a MAS is updated each time the LLC is accessed. Membership is tested in every bloom filter for each new MAS and when a

MAS belonging to one of the filters is found the LLC is immediately reconfigured to the corresponding size. If the MAS is found in multiple filters the smaller size is preferred.

5.6 The Artificial Neural Network Algorithm

As with the BLOOM algorithm, the Artificial Neural Networks procedure (ANN) described in Algorithm 4 also uses the mapping of MAS to optimal LLC configurations generated by the corpus of training applications. The ANN algorithm trains a neural network to recognize MAS and determine which LLC configuration will be optimal. We used the FANN library [18] to implement artificial neural networks in the training phase and integrated this library with Multi2Sim to control LLC sizing during execution.

The ANN algorithm uses a single neural network with one input neuron for each bit in the MAS and one output neuron for each LLC configuration. These output neurons form the configuration vector which will select which configuration is optimal given an observed MAS. Each bit in this vector corresponds to a distance hardware configuration, so for the output to be valid exactly one bit in the vector must be set and the others must all be zero, otherwise the selected configuration is ambiguous.

Profiling data from the training corpus associates each unique observed MAS with an optimal LLC configuration vector as determined by the cost function described in Section 6.3. During the training phase, only perfect matches between input MAS vector and output configuration vectors are considered to be successfully learned. Algorithm 4 uses the hamming_weight function from Equation (1) which is defined as the sum of all the bits in its argument.

The neural network is taught to learn a function which maps a MAS to a vector of bits with one bit for each hardware configuration. In the training data, all bits of the output vector are zero except for the index corresponding to the optimal LLC configuration. Sections 6.2 and 6.3 give more details regarding the ANN implementation.

While the candidate software executes, a MAS is updated each time the LLC is accessed. After a window of execution, as defined in Section 4.1, the new MAS is run through the neural network to obtain an output vector of one bit for each LLC configuration. If none of the bits in the output vector are 1 or if multiple bits are 1 the output is considered to be low confidence and no action is taken. If there is exactly one bit set to 1 in the output vector then the LLC is immediately reconfigured to the size corresponding to that index.

In the next section, we will describe the details of gathering profiling data from the training applications.

6. Gathering Profiling Data

We gather profiling data from four simple programs we have written. The reason we picked these programs is that they each one of them contains basic operations that are used in many larger and more sophisticated applications.

6.1 The Application Corpus

The profiling data used to train the cache size selection algorithms described in Section 5 comes from an application corpus we developed of four small programs listed in Table 1. Each application is focused on a single function and can be executed with input data of three sizes.

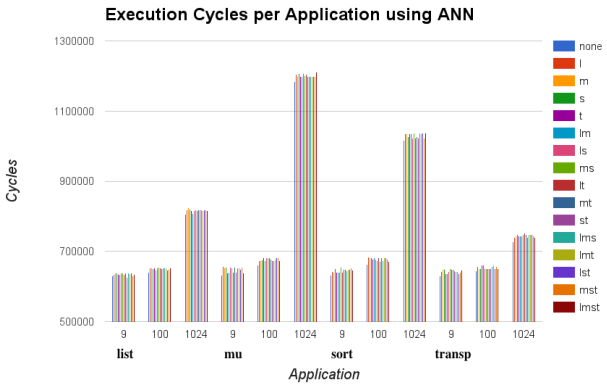

Figure 1: Application runtime for our training programs when executed with the ANN algorithm using a neural network trained on profiling data from each subset of our application corpus.

As a small test, Figure 1 shows how application execution time is affected by training the ANN function on different permutations of data from our training corpus. The legend in the figures uses a single letter to indicate each application. The letter "l" indicates the "list" application, "m" is the "mul" application, "s" the "sort" application, and "t" the "transp" application; for example, the bars labeled "lms" trained the neural network on profiling data from the list, mul, and sort applications but not the transp application. The bars labeled "none" use the full LLC size without the ANN algorithm. In each case, the execution time is primarily affected by the application and data size. Using profiling data from more or fewer applications does not affect the run time of the application or the performance of the ANN though it will affect the overall efficiency of each application execution. That is, getting same execution time but with smaller cache.

6.2 Signature Selection

Each application from the corpus was run at each data size and during each execution the Memory Access Signatures was recorded after each LLC access. This produced 12 sets of MAS. All MAS appearing in multiple sets were considered non-predictive and removed from the training data, leaving 12 sets of globally unique MAS each one observed during the execution of a single application on a single data size. Each set of MAS was then associated with a hardware configuration and this mapping from MAS to optimal LLC size was used to populate the bloom filters in Algorithm 3 and the neural network in Algorithm 4.

6.3 Associating a Signature with an LLC Size

The cache size selection algorithms described in Section 5 all require a finite number of hardware configurations to choose among. Theoretically, these configurations could be as fine grained as a single cache block or row. For these experiments, we have chosen to select among 5 different LLC sizes. We chose these sizes to be 20%, 40%, 60%, 80%,

and 100% of the maximum LLC size which is 16K blocks of 64 bytes each (1024 KB). Each of the 12 application and data size pairs from our training corpus was executed with LLC fixed at each of these 5 levels. The method requires a cost function to evaluate which configuration is most efficient when building a profiling model. In this experiment, we chose a cost function which minimized the LLC size without increasing the execution time. The set of MAS associated with the optimal application and data size was then mapped in the profiling model to the smallest LLC size that did not increase the execution time as measured in simulated CPU cycles. Now we will use the profile data gathered to affect the efficiency of different programs that were not profiled before.

7. Experimental Results

We used benchmarks from the PARSEC 3.0 [16] benchmark suite to evaluate the efficacy of our LLC size selection method. Figure 2 shows the LLC size selected by the EWSS, BLOOM, and ANN algorithms over the course of executing the blackscholes benchmark from the PARSEC suite.

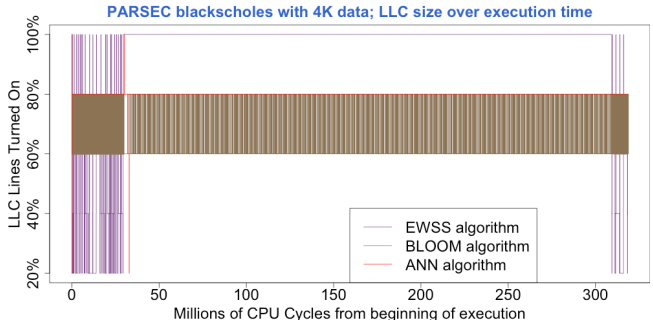

Figure 2: The function of LLC size over time as determined by the EWSS, BLOOM, and ANN algorithms. The stair-step nature of the EWSS algorithm can be clearly observed near the start of execution as it iterates through all configurations after each program phase change.

As shown in Figure 2, the ANN and BLOOM algorithms alternate between powering off 20% and 40% of the LLC for the majority of the execution period (although the BLOOM algorithm alternates much more frequently). Due to working set instability, the WSS algorithm barely performs better than operating without any hardware adjustments.

In this example, the ANN algorithm powered an average of 80% of the LLC, the BLOOM algorithm powered 61% of the LLC on average, and EWSS algorithm powered on average 94% of the LLC. At the same time, the LLC miss rate was 2.8% while using EWSS, 3.1% while using BLOOM, and 3.0% when using ANN. However, there was less that 1% difference in execution time among the three algorithms. This makes ANN and BLOOM an improvement of 15% and 35% in LLC cache size respectively at the cost of only ¼ percentage points additional miss rate.

We performed this analysis on each benchmark from the PARSEC suite and the results are summarized in Figures 3 through 5. Figure 5 shows the improvements provided by our method through reduced LLC size during application execution. The BLOOM algorithm turned off an average of 38% of the LLC while the EWSS algorithm only turned off 23% of the LLC. The ANN algorithm performed slightly worse than EWSS by turning off an average of 14% of the LLC.

Though they both use the same training corpus, the ANN algorithm does not select as small an average cache size as the BLOOM algorithm. One reason for this is that the ANN model is not trained on as many MAS as the BLOOM model is. Duplicate MAS observed across multiple profiles in the training corpus are removed from the training data before FANN creates a neural network. This step is to prevent very common MAS from being mapped to two different optimal LLC sizes, which would prevent the model from representing a function from MAS to optimal LLC size and hence would make the relationship unlearnable by the neural network. This step is unnecessary in the BLOOM algorithm since each bloom filter is disjoint; if the MAS is found in multiple filters the smallest LLC size can always be preferred. Figure 6 shows another informative result relating to the performance repercussions of the amount of data used when training the ANN profiling model.

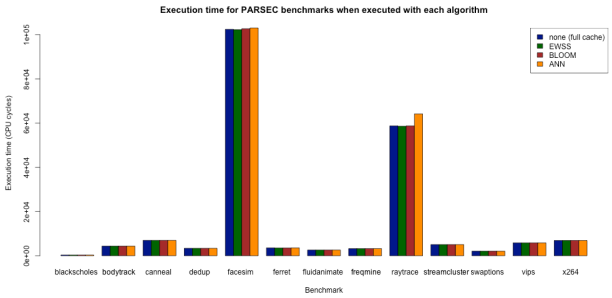

Figure 3: Execution time of benchmarks from the PARSEC suite using the LLC size selection algorithms presented in this paper. The total number of cycles is not significantly affected by the algorithms.

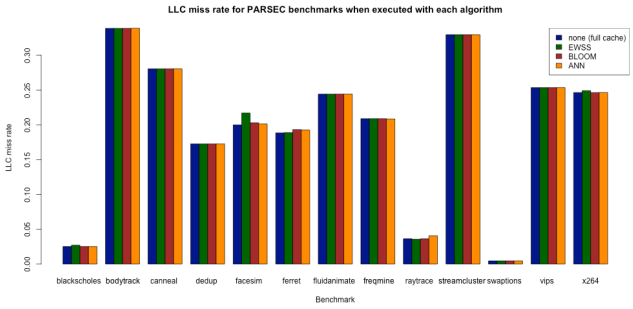

Figure 4: LLC miss rate of benchmarks from the PARSEC suite using the size selection algorithms presented in this paper. Reducing the cache size does result in a small increase in the frequency of a cache miss. An average increase of 0.30% was observed in the BLOOM algorithm and 0.38% in the ANN algorithm when compared to the full cache size.

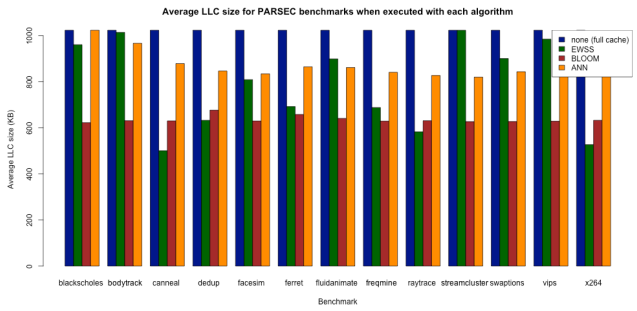

Figure 5: The last level cache size averaged over each CPU cycle observed while executing benchmarks from the PARSEC suite while using each LLC size selection algorithm presented in this paper.

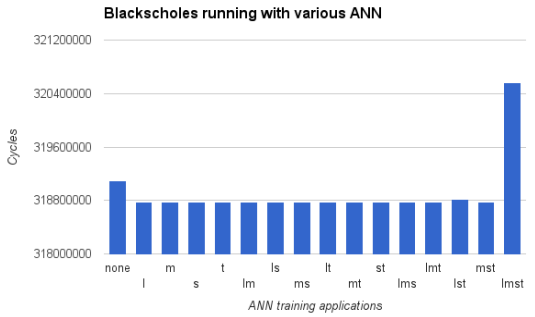

Figure 6: Runtime of PARSEC blackscholes benchmark when executed on 4K data with the ANN algorithm using a neural network trained on profiling data from each subset of our application corpus.

The labels on the horizontal axis of Figure 6 are the same as those in the legend of Figure 1. This figure clearly shows a much larger variance in the execution time of the ANN algorithm when trained on profiling data from all 4 applications. This could indicate that the neural network began to overfit the model when exposed to too many examples. Conversely, it could also indicate that too many data points lead to a flatness problem that hindered the ability of the neural network to learn a good partition of the function range. We did not have time to investigate the cause of this variance, so it is left as a question for future research.

8. Hardware cost

To implement the proof-of-concept technique for reducing LLC size presented in this paper some additional hardware costs will be incurred. The LLC requires 2 extra bits per cache block to keep track of read and write accesses so it can be efficiently reconfigured. The LLC will also need 8 bits to maintain the MAS which will need to be updated on each access to the LLC.

The BLOOM algorithm will require one bloom filter for each supported cache size and the number of bits required in each the filter will grow with the size of the training corpus (to accommodate more MAS at the same error rate). To have a 1% chance of collisions will require a set of bloom filters with 4 bits per MAS. In our experiments we added an average of 16,114 MAS into five bloom filters. This would necessitate 40KB of bloom filter memory. The BLOOM algorithm disabled an average of 200KB in our LLC resulting in a total efficiency 160KB of unpowered cache memory. A bloom filter in this configuration will can operate at 5 kHz [24], but our size selection algorithms never block the execution pipeline.

The ANN algorithm requires a neural network with 64 input bits (equal to the length of a MAS) and one output bit for each LLC size. An integrated circuit implementing the neural network will also be required by the ANN algorithm. This neural network is proportional to the number of hardware configurations. In our experiment, we used a fully connected 3 layer neural network with 64 input neurons, 32 hidden neurons, and 5 output neurons. This configuration requires 10 KB of additional memory. The ANN algorithm disabled an average of 46KB in our LLC resulting in a total efficiency 36KB of unpowered cache memory.

9. In General

We are presenting a general method for using profiling data from a limited corpus of applications to improve performance across all unseen software. Our proof-of-concept implementation of this technique exemplifies the steps needed to apply this method to any domain.

- Start with a set of available profiling data. Or generate profiling data from a small corpus of applications, or you can use our presented list.
- Determine a fixed set of states into which the system will be configurable
- Define a cost function to minimize over the profiling data
- Design a suitable execution signature
- Implement a data model and algorithm to map those signatures to the optimal configuration

If you already have old profiling data, then you can extract the data needed for your criteria. **As more profiling data become available, you can refine the learning scheme**.

In the case of our example, we wrote a corpus of 4 small applications and chose 5 sizes of LLC as the system configurations. The cost function was defined to select the smallest LLC size that did not increase cache misses. We designed memory access signatures to characterize the software/hardware interaction pattern and used MAS to implement a BLOOM and ANN algorithm which predicted optimal cache size using a set of bloom filters or an artificial neural network respectively.

This technique is broadly useful to any environments where novel software is regularly executed. This includes Operating Systems where application software can be downloaded and executed only a single time as well as data centers cannot always guarantee the hardware platform on which an application will execute. Environments using hypervisors or containers to provide PaaS and IaaS would like to completely hide the underlying hardware platform from the application software. This method allows the the provider to mitigate the performance cost of running an application on a hardware platform for which it was not designed without incurring the overhead of profiling the application directly.

The main reason we believe the proposed idea is applicable to many other situations beside LLC is that the interaction of hardware and software is based on patterns not programs. A specific branch predictor for example does well when the branches follow a specific pattern. A cache replacement algorithm with specific

parameters works well with the access follows a specific patterns. Even **when multiple treads are executing in parallel**, what they cumulatively generate is a pattern. In the proof-of-concept presented in this paper, the LLC sees access pattern, regardless of whether these patterns are generated by one thread or multiple threads. Therefore, **our proposed method strives to capture patterns.**

10. Future Work

This paper is a first step proof-of-concept. There are several other questions to be answered.

- What is the minimum, and type, of training applications after which we see diminishing return in performance?
- Is the answer of the above question dependent on the structure we are trying to configure? If yes, in what way?
- Can the scheme proposed in this paper be combined with the dynamic techniques to achieve even better results?
- If we want to reconfigure different structures in the processor, can we have a generalized module to reconfigure them all?

We also plan to try our technique in OS scheduling as well as compiler optimizations.

References

[1] Denning, Peter J. "Working sets past and present." *Software Engineering, IEEE Transactions on* 1 (1980): 64-84.

[2] Dhodapkar, Ashutosh S, and James E Smith. "Managing multi-configuration hardware via dynamic working set analysis." *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on* 2002: 233-244.

[3] Balasubramonian, Rajeev et al. "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures." *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture* 1 Dec. 2000: 245-257.

[4] Zhang, Chuanjun, Frank Vahid, and Roman Lysecky. "A self-tuning cache architecture for embedded systems." *ACM Transactions on Embedded Computing Systems (TECS)* 3.2 (2004): 407-425.

[5] Gove, Darryl. "CPU2006 working set size." *ACM SIGARCH Computer Architecture News* 35.1 (2007): 90-96.

[6] Han, Tianyi David, and Tarek S Abdelrahman. "Automatic tuning of local memory use on GPGPUs." *arXiv preprint arXiv:1412.6986* (2014).

[7] Ipek, Engin et al. "Efficient architectural design space exploration via predictive modeling." *ACM Transactions on Architecture and Code Optimization (TACO)* 4.4 (2008): 1.

[8] Abella, Jaume et al. "IATAC: a smart predictor to turn-off L2 cache lines." *ACM Transactions on Architecture and Code Optimization (TACO)* 2.1 (2005): 55-77.

[9] Banerjee, Subhasis, and SK Nandy. "Program phase directed dynamic cache way reconfiguration for power efficiency." *Proceedings of the 2007 Asia and South Pacific Design Automation Conference* 23 Jan. 2007: 884-889.

[10] Adve, Sarita V et al. "Changing interaction of compiler and architecture." *Computer* 30.12 (1997): 51-58.

[11] Dhodapkar, Ashutosh S, and James E Smith. "Comparing program phase detection techniques." *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture* 3 Dec. 2003: 217.

[12] Yuan, Pengfei, Yao Guo, and Xiangqun Chen. "Experiences in profile-guided operating system kernel optimization." *Proceedings of 5th Asia-Pacific Workshop on Systems* 25 Jun. 2014: 4.

[13] Schulte, Eric et al. "Post-compiler software optimization for reducing energy." *ACM SIGARCH Computer Architecture News* 24 Feb. 2014: 639-652.

[14] Chen, Dehao et al. "Taming hardware event samples for FDO compilation." *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization* 24 Apr. 2010: 42-52.

[15] Sherwood, Timothy, and Brad Calder. "Time varying behavior of programs." 100.101 (1999): 0.5.

[16] Bienia, Christian et al. "The PARSEC benchmark suite: Characterization and architectural implications." *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* 25 Oct. 2008: 72-81.

[17] Ubal, R et al. "Multi2sim: A simulation framework to evaluate multicore-multithread processors." *IEEE 19th International Symposium on Computer Architecture and High Performance computing, page (s)* 2007: 62-68.

[18] Nissen, Steffen. "Implementation of a fast artificial neural network library (fann)." *Report, Department of Computer Science University of Copenhagen (DIKU)* 31 (2003).

[19] Smith, Michael D. "Overcoming the challenges to feedback-directed optimization (Keynote Talk)." *ACM SIGPLAN Notices* 1 Jan. 2000: 1-11.

[20] Dhodapkar, Ashutosh S, and James E Smith. "Saving and restoring implementation contexts with co-designed virtual machines." Workshop on Complexity-Effective Design 30 Jun. 2001.

[21] Bloom, Burton H. "Space/time trade-offs in hash coding with allowable errors." *Communications of the ACM* 13.7 (1970): 422-426.

[22] Smith, James E. "A study of branch prediction strategies." *Proceedings of the 8th annual symposium on Computer Architecture* 12 May. 1981: 135-148.

[23] Zhang, Weihua et al. "Multilevel Phase Analysis." *ACM Transactions on Embedded Computing Systems (TECS)* 14.2 (2015): 31.

[24] Lyons, Michael J., and David Brooks. "The design of a Bloom filter hardware accelerator for ultra low power systems." *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*. ACM, 19 Aug 2009. 371-376