Synthesizing Contention

Jason Mars
University of Virginia
jom5x@cs.virginia.edu

Mary Lou Soffa
University of Virginia
soffa@cs.virginia.edu

ABSTRACT

Multicore microarchitecture designs have become ubiquitous in today's computing environment enabling multiple processes to execute simultaneously on a single chip. With these new parallel processing capabilities comes a need to better understand how co-running applications impact and interfere with each other. The ability to characterize and better understand *cross-core performance interference* can prove critical for a number of application domains, such as performance debugging, compiler optimization, and application co-scheduling to name a few. We proposed a novel methodology for the characterization and profiling of cross-core interference on current multicore systems, which we call *contention synthesis*. Our profiling approach characterizes an applications *cross-core interference sensitivity* by manufacturing contention with the application and observing the impact of this *synthesized* contention on the application.

Understanding how to synthesize contention on current chip microarchitectures is unclear as there are a number of potentially contentious data access behaviors. This is further complicated by the fact that current chip microprocessors are engineered and tuned to circumvent the contentious nature of certain data access behaviors. In this work we explore and evaluate five designs for a contention synthesis mechanism. We also investigate how these five contention synthesis engines impact the performance of 19 of the SPEC2006 benchmarks on two state of the art chip multiprocessors, namely Intel's Core i7 and AMD's Phenom X4 architectures. Finally we demonstrate how contention synthesis can be used to accurately characterize an application's *cross-core interference sensitivity.*

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.3.4 [**Programming Languages**]: Processors—*run-time environments, compilers, optimization, debuggers*; D.4.8 [**Operating Systems**]: Performance—*measurements, monitors*

General Terms

Performance, Contention, Multicore

Keywords

cross-core interference, profiling framework, program understanding

1. INTRODUCTION

Multicore architectures are quickly becoming ubiquitous in today's computing environment. With each new generation of general purpose processors, much of the performance improvement in micro-architectural design is typically achieved by increasing the number of individual processing cores on a single chip. However, shared on-chip resources and the memory subsystem is typically shared among many cores, resulting in a potential performance bottleneck when scaling up multiprocessing capabilities.

Current multicore architectures include early levels of small core-specific private caches, and larger caches shared among multiple cores [8]. When multiple processes or threads run in tandem, they can contend for the shared cache by evicting a neighboring process or thread's data in order to cache its own data. This form of contention occurs when the working set of the neighboring processes or threads exceed the size of the private caches, relying on the shared cache resources. This contention can result in a significant degradation in application performance.

When an application suffers a performance degradation due to contention for shared resources with an application on a separate processing core, we call this *cross-core interference*.

The ability to characterize an application's sensitivity to cross core interference can prove indispensable for a number of applications. Compiler and optimization developers can use knowledge about the contention sensitivity of a particular code region to develop both static and dynamic *contention conscious* compiler techniques and optimization heuristics. For example, knowledge of contention sensitive code regions can be used to direct where software cache prefetching should be applied. Software developers can also use this cross-core interference sensitivity information for performance debugging and software tuning. The ability to characterize the most contention sensitive application phases allows the programmer to pin-point parts of the application on which to focus. In addition to performance debugging, software tuning, and compiler optimization, the characterization of application sensitivity to cross core interference

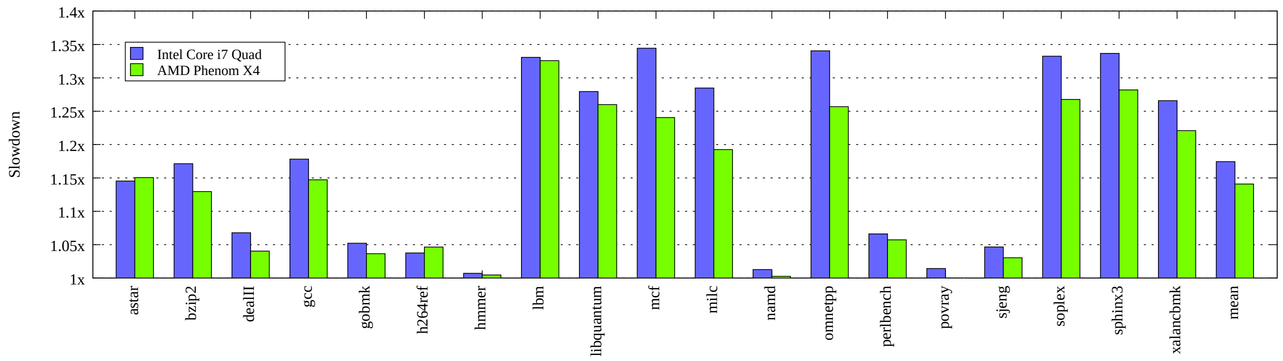

Figure 1: Performance impact due to contention from co-location with LBM.

can enable smarter, *contention conscious* dynamic and online scheduling techniques. For example, the understanding of an application's contention sensitivity characteristics enables contention conscious application co-scheduling. Applications that have higher demands on shared memory resources can be co-located with applications that have a lower demand to gain better performance and throughput.

While ad-hoc and indirect approaches, such as measuring cache hits and misses via performance counters, can give a coarse indication of cross-core interference sensitivity, they are not sufficient to provide accurate and detailed profiling information about on-chip contention. Monitoring the shared cache misses directly is not sufficient in that not all cache misses reported by the hardware performance monitoring unit are misses that relate to code dependencies. In modern processors many misses reported by the hardware monitors are caused by hardware prefetches and hardware page table walks [8]. These effects do not relate to contention and are indistinguishable from cache misses that do.

We propose a cross-core interference profiling environment that uses *contention synthesis*. To accurately characterize and profile an applications sensitivity to cross-core interference, we *synthesize* contention, meaning we synthetically create contention with the host application. This contention synthesis is achieved by synthetically applying pressure on the shared cache using a *contention synthesis engine* (CSE). The profiling framework manipulates the execution of the CSE while observing the effect on the host application, measuring the impact over time, and assigning an impact score to the application. However, understanding how to synthesize contention on current chip microarchitectures is unclear as there are a number of differing contentious data access behaviors in addition to the fact that current chip microprocessors are engineered and tuned to circumvent the poor cache performance of certain data access behaviors.

In this work we explore the design space of our contention synthesis engine and investigate how contention synthesis can be used to characterize cross-core performance interference on modern multicore architectures. We have designed and evaluated five contention synthesis mechanisms that mimic five common data access behaviors. These include the random access of elements in a large array, the random traversal of large linked data structures, a real world fluid dynamics application (the `lbm` SPEC2006 benchmark), data movement in 3D object space commonly found in simulations and scientific computing, and finally, a contention synthesis engine that was constructed by reverse engineering *lbm*, finding its most contentious code, and further tweaking it to construct a highly contentious synthesis engine. In addition to presenting the design and implementation of these contention synthesis methods we investigate how these five contention synthesis engines impact the performance of 19 of the SPEC2006 benchmarks on two state of the art chip multiprocessors, Intel's Core i7 and AMD's Phenom X4 architectures. We also answer a number of questions as to whether the cross-core interference properties of applications tend to remain consistent regardless of the particular contention synthesis method chosen. Finally we demonstrate how contention synthesis can be used to accurately characterize an applications *cross-core interference sensitivity*.

The contributions of this work includes:

- The discussion of a novel methodology for the characterization of cross-core performance interference on current multicore architecture.
- The design and implementations of five contention synthesis mechanisms.
- The evaluation and study of the impact these five contention synthesis mechanism on 19 SPEC2006 benchmarks on both the Intel Core i7 and AMD Phenom X4 Architectures.
- The *cross-core interference sensitivity* scoring of the SPEC2006 benchmarks.

Next in Section 2 we motivate our work. We then provide an overview of our profiling approach in Section 3. Section 4 presents our contention synthesis methodology. We evaluate our contention synthesis approach in Section 5. Section 6 presents related work, and finally, we conclude in Section 7.

2. MOTIVATION

With the recent growth in popularity of multicore architecture, comes an increase in the parallel processing capabilities of commodity systems. These commodity systems are now common-place both in the general purpose desktop and laptop markets as well as in industry data-center and computing clusters. Companies such as Google, Yahoo, and Microsoft use these off the shelf computing components to build their data-centers as they are cheap, abundant and easily replaceable [3]. The increase in parallel processing capabilities in these chip architectures are in fact leading to server consolidation. However, the memory wall is preventing these parallel processing capabilities from being fully realized.

The memory subsystem on current commodity multicore architectures is shared among the processing cores. Two representative examples of the state of the art multicore chip designs are the Intel Core i7 Quad Core chip and AMD's Phenom X4 Quad Core. Intel's Core i7 has four processing cores, each with a private 32kb L1 cache and a 256kb L2 cache. A large 8mb L3 cache is shared among the four cores [8]. AMD's Phenom X4 also has 4 cores with a similar cache layout. Each core has a private 64kb L1 and 512kb L2, with a shared 6mb L3 cache. These chips were designed to accommodate 4 simultaneous streams of execution. However, as we can see through experimentation, their shared caches and memory subsystem often cannot efficiently accommodate even 2 co-running processes.

Figure 1 illustrates the potential *cross-core interference* that can occur when multiple co-running applications are executing on current multicore architectures. We perform the following experiment using the Core i7 and Phenom X4 architectures. In this experiment we study the cross-core performance interference caused to each of the SPEC2006 benchmarks when co-running with `lbm` (one of the SPEC2006 benchmarks known be especially heavy on the on-chip memory subsystem). Figure 1 shows the slowdown of each benchmark due to the cross-core interference. Each application was executed to completion on their `ref` inputs. On the y-axis we show the execution time of the application while co-running with `lbm` normalized to the execution-time of the application running alone on the system. The first bar in Figure 1 presents this data for the Core i7 architecture, and the second bar for the Phenom X4. As this graph shows, there are severe performance degradations due to cross-core interference on a large number of Spec benchmarks. The large last level on-chip caches of these two architectures do little to accommodate many of these co-running applications. On a number of benchmarks including `lbm`, `mcf`, `omnetpp`, and `sphinx`, this degradation approaches 35%.

In addition to the general performance degradation, this *sensitivity* to cross-core interference is particularly undesirable for real time and latency sensitive application domains. In the latency sensitive domain of web search for instance this kind of cross core interference can cause unexpected slowdowns, negatively impacting the QoS on a search query. A commonly used solution is to simply disallow the co-location of latency sensitive applications with others on a single machine, resulting is lowered utilization and higher energy cost [14].

Note that not all applications are effected by the contention properties of their co-runners. Applications such as `hmmer`, `namd`, and `povray` seem immune to `lbm`'s cross core interference. This observation shows that cross-core interference sensitivity vary substantially across applications.

3. PROFILING FRAMEWORK

Contention and cross-core interference can only occur dynamically, and depends on a combination of the application's memory behavior, the design of the particular underlying architecture, and the applications co-running on this microarchitecture at any particular time. Because of these properties, characterizing this sensitivity using static analyses is intractable. Also, the traditional profiling of the application in isolation is not sufficient as no contention is actually occurring. Although it is possible to observe the performance counters of current architecture designs to analyze cache misses, etc, these are indirect means to infer an application's memory behavior as no actual contention or cross-core interference is occurring. Our methodology is to profile the application with real contention in a controlled environment, where we manufacture contention using a *contention synthesis* mechanism and dynamically monitor and profile the impact on the host application.

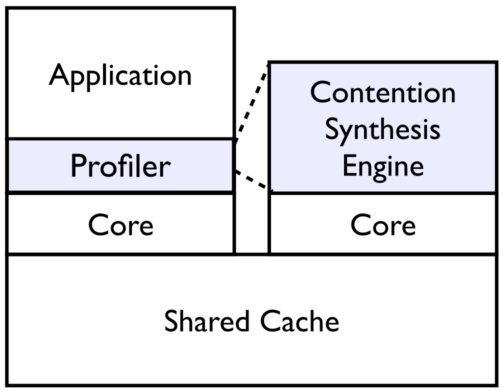

Figure 2: Our Profiling Framework

Figure 2 shows an overview of our *cross core interference profiling environment.* This figure shows a multicore architecture with two separate cores sharing an on-chip cache and memory subsystem. The shaded boxes show our profiling framework, which is composed of the profiler runtime and a *contention synthesis engine* (CSE). As shown on the left side of Figure 2, the host application is controlled by the profiler runtime and is monitored throughout the execution of the application. Before the execution of the host application, the profiler spawns the *contention synthesis engine* on a neighboring core, as shown to the right of the figure. This CSE shares the cache and memory subsystem of the host application. As the application executes, the CSE engine aggressively accesses memory trying to cause as much cross-core interference as possible. The profiler manipulates the execution of the contention synthesis engine allowing bursts of execution to occur. Slowdowns in the application's instruction retirement rate that result from this bursty execution are monitored using the *hardware performance monitoring* (HPM) information [8]. This intermittent control of the CSE and monitoring of the HPM is achieved using the *periodic probing* approach [15]. A timer interrupt is used to periodically execute the monitoring and profiling directives. This has shown to be a very low overhead approach for the dynamic monitoring and analysis of applications.

4. SYNTHESIZING CONTENTION

Many types of applications cause cache contention. With the continuing advances in micro-architectural design simply accessing a large amount of data does not necessarily imply high pressure on cache and memory performance. The type of data access pattern and the way that data is mapped into the cache is very important to consider when constructing the contention synthesis engine. Structures such as hardware cache prefetchers and victim caches can avert poor and contentious cache behavior even when the working set of the application is very large. The features and functionality of these hardware techniques are difficult to anticipate as vendors keep these details closely guarded. Access patterns that exhibit a large amount of spatial or temporal locality can easily be prefetched into the earlier and later levels of cache.

4.1 Designing Contention Synthesis

To design our contention synthesis engine we explored and experimented with a number of common data access patterns. These designs consist of the random access of elements in a large array, the random traversal of large linked data structures, a real world fluid dynamics application (the `lbm` SPEC2006 benchmark), data movement in 3d object space commonly found in simulations and scientific computing and finally, and finally we reverse engineered *lbm*, found its most contentious code, and further tweaked it to construct a highly contentious synthesis engine we call ”The Sledgehammer.”

4.1.1 Naive

Figure 3 shows the `C` implementation of our naive contention synthesis mechanism.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

char *data;

main(int argc, char *argv[]) {
  srand(time(0)+getpid());
  if(argc<2) exit(1);
  int bytes=atoi(argv[1])*1024;
  data=(char*)malloc(bytes);
  for(int i=0; i<bytes; i++) data[i]=rand()%256;

  while(1) {
    for(int j=0; j<bytes-2; j++) {
      data[rand()%bytes]+=data[rand()%bytes];
    }
  }
  printf("%d\n",(int)data[rand()%bytes]);
}
```

Figure 3: Naive Contention Synthesis

The first inclination is to simply access a large array of memory (a little larger than the L3 cache) performing both loads and stores. Our earliest CSE design attempts consisted of an array of memory just larger than the last level of on-chip cache, which we traversed in a number of clever ways. However, the hardware prefetchers on both the Intel and AMD chips cleverly prefetched to early levels of cache. One example of an approach subverted by the hardware prefetchers was the caching of 10,000 random numbers to be used to access the elements of a large array. This naive design evolved to simply calculating the random index on the fly as shown in Figure 3. The hardware prefetcher was unable to anticipate these memory accesses. The drawback of this approach however is the fact that each memory access is interleaved with the logic to calculate the random number, allowing for a high degree of instruction level parallelism.

4.1.2 Linked Data Structure

Figure 4 shows the `C++` implementation of our linked data traversal contention synthesis mechanism based on a binary search tree.

This design for the CSE consisted of the random construction and traversal of a binary search tree. There were also

```
#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;

const int payload_size=128;

void gen_name(char *ret) {
  for(int i=0; i<payload_size; i++) {
    ret[i]=(char)rand()%256;
  }
}

struct tree_node {
  ~tree_node(){
  if(left) delete left; if(right) delete right;
  }
  tree_node* left;
  tree_node* right;
  int data;
  char text[payload_size];
};

class BST {
  private:
    tree_node* root;
  public:
    BST() {
      root = NULL;
    }
    bool isEmpty() const { return root==NULL; }
    void insert(int);
    void remove(int);
    void clear(){if(root){delete root;} root=NULL;}

    unsigned long trample();
    unsigned long trample(tree_node *p);
};

...[standard implementation of insert and remove]

unsigned long BST::trample(){return trample(root);}
unsigned long BST::trample(tree_node *p) {
  unsigned long ret=0;
  if(p != NULL) {
    //Using random traversal + 5%
    if(p->data%2) { //Using p->data instead of rand + 2%
      if(p->left) ret+=trample(p->left);
      if(p->right) ret+=trample(p->right);
      ret+=(unsigned long)p->text[p->data%payload_size];
      p->data+=ret; //Moding data + 6%
      p->text[p->data%payload_size]=p->data%256;
    }
    else {
      if(p->right) ret+=trample(p->right);
      if(p->left) ret+=trample(p->left);
      ret-=(unsigned long)p->text[p->data%payload_size];
      p->data+=ret; //Moding data + 6%
      p->text[p->data%payload_size]=p->data%256;
    }
  }
  return ret;
}

int main(int argc, char *argv[]) {
  int footprint=8192;

  BST b;
  srand(time(0)+getpid());

  unsigned int node_size=sizeof(tree_node)+sizeof(BST);

  for(int i=0; i<footprint*1024/node_size; i++) {
    b.insert(payload_size+(rand()-payload_size));
  }

  unsigned long long sum=0;
  while(1)
     sum+=b.trample()+b.trample();
}
```

Figure 4: Contention Synthesis Using a Linked Data-structure

a number of steps taken to reverse optimize (de-optimize for poor performance) this linked structure CSE approach. For example, the `trample` function is a specialized traversal that recursively picks whether the left or right subtree is to be *trampled* first. In the final design of this CSE, each tree node consisted of an `id` and a `payload`, and this custom traversal function, `trample`, is used, as shown in Figure 4 The payload consisted of a number of random bytes (128 in our design) to have the node map into its own cache line. The contentious kernel of this approach uses the `trample` function to performed a random depth first search through the tree touching and changing the data alone the way.

4.1.3 LBM from SPEC2006

The implementation of the LBM benchmark can be found in the official SPEC2006 benchmarks suite [7]. LBM is an implementation of the "Lattice Boltzmann Method" (LBM). The Boltzmann Method is used to simulate incompressible fluids. We selected this benchmark as one of our synthesis mechanisms, as it proved to be one of the most contentious of the SPEC2006 benchmark suite. For a complete description of LBM please refer to [7].

4.1.4 3D Data Movement

Figure 5 shows the C++ implementation of our 3D data movement contention synthesis mechanism.

```
#include <iostream>
#include <cstdlib>

using namespace std;

const int nug_size=128;

class nugget {
  public:
    char n[nug_size];
    nugget(){
      for(int i=0; i<nug_size; i++){n[i]=rand()%256;}
    }
};

class block {
  public:
    nugget ***b;
    unsigned size;
    block(unsigned sz);
    ~block();
};

block::block(unsigned sz) {
  b=new nugget**[sz];
  for(int i=0; i<sz; i++) {
    b[i]=new nugget *[sz];
    for(int j=0; j<sz; j++)
      b[i][j]=new nugget[sz];
  }
  size=sz;
}

block::~block() {
  for(int i=0; i<size; i++) {
    for(int j=0; j<size; j++)
      delete [] b[i][j];
    delete [] b[i];
  }
  delete [] b;
}

int main() {
  const int size=30;
  block b1(size);
  block b2(size);
  block b3(size);

  cout << "smash" << endl;
  while(1)
    for(int i=0; i<size; i++)
      for(int j=0; j<size; j++)
        for(int k=0; k<size; k++)
          b1.b[i][j][k]=b2.b[j][k][i]=b3.b[i][j][k];
  return 0;
}
```

Figure 5: Blockie

This 3D data movement micro benchmark consists of a number of large 3D arrays of double precision values that represent solid virtual cubes. In our experimentation the dimensionality chosen for these cubes was 30x30x30. The contentious kernel of this CSE is the transposition of cells of each cube into the space of another cube. The cells of one cube is continuously copied to another.

4.1.5 "The Sledgehammer"

Figure 6 shows the C implementation of our sledgehammer contention synthesis mechanism.

```
#include <stdlib.h>

typedef double LBM_Grid[26000000];

static double *srcGrid,*dstGrid;
int main()
{
  const unsigned long margin = 400000,
    size = sizeof( LBM_Grid ) + 2*margin*sizeof( double );
  srcGrid = malloc( size );
  dstGrid = malloc( size );
  srcGrid += margin;
  dstGrid += margin;

  while(1)
  {
    int i;
    for( i = 0; i < 26000000; i += 20 ) {
      dstGrid[i] = srcGrid[i];
      dstGrid[i-1998] = srcGrid[(1)+i];
      dstGrid[i+2001] = srcGrid[(2)+i];
      dstGrid[i-16] = srcGrid[(3)+i];
      dstGrid[i+23] = srcGrid[(4)+i]
      dstGrid[i-199994] = srcGrid[(5)+i];
      dstGrid[i+200005] = srcGrid[(6)+i];
      dstGrid[i-2010] = srcGrid[(7)+i];
      dstGrid[i-1971] = srcGrid[(8)+i];
      dstGrid[i+1988] = srcGrid[(9)+i];
      dstGrid[i+2027] = srcGrid[(10)+i];
      dstGrid[i-201986] = srcGrid[(11)+i];
      dstGrid[i+198013] = srcGrid[(12)+i];
      dstGrid[i-197988] = srcGrid[(13)+i];
      dstGrid[i+202011] = srcGrid[(14)+i];
      dstGrid[i-200002] = srcGrid[(15)+i];
      dstGrid[i+199997] = srcGrid[(16)+i];
      dstGrid[i-199964] = srcGrid[(17)+i];
      dstGrid[i+200035] = srcGrid[(18)+i];
    }
  }
  return 0;
}
```

Figure 6: Sledge

This last design is the result of reverse engineering and investigating lbm to learn its contentious core nature. This name is motivated by the fact that the behavior of this design can be visualized as hitting an element in a 1D or 2D array, and a number of sparsely surrounding elements feel the shock-wave, e.g. are effected. As shown in Figure 6, the final version of this CSE first allocates two large arrays and enters its contentious kernel which copies data back and forth with this sledgehammer pattern.

5. EVALUATION

First we evaluate the effectiveness of these five contention synthesis engines at generating contention, and investigate how these varying contention generation mechanisms affect real applications on current commodity multicore microarchitectures. Our second goal is to use contention synthesis to characterize cross-core interference sensitivity and evaluate the ability of such a profiling framework to accurately identify applications that are indeed sensitive to cross core interference.

5.1 Evaluating Contention Synthesis Designs

With the variety of contention synthesis mechanisms presented above, a number of questions arise. The first has to do with whether there is a drastic difference between the interactions of different applications to the different contention syntheses designs. We hypothesize that contention is agnostic to the nature of the memory access. We seek to evaluate this very question. The other goal of this evaluation is to learn whether there exist a synthesis engine that is better than all others, and if so, to identify it.

Figures 7 and 8 show the performance impact of co-running each of the contention synthesis designs with each of the SPEC2006 benchmarks (C/C++ only, run to completion on

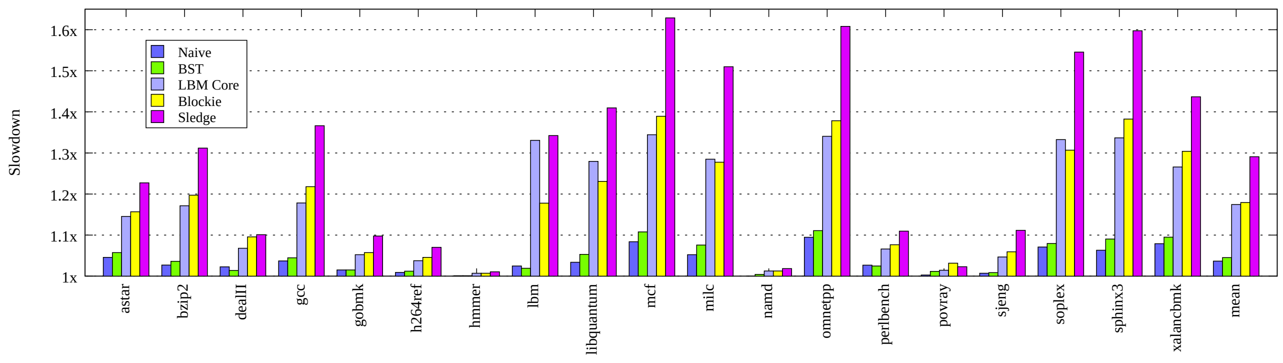

Figure 7: Slowdown caused by contention synthesis on Intel Core i7.

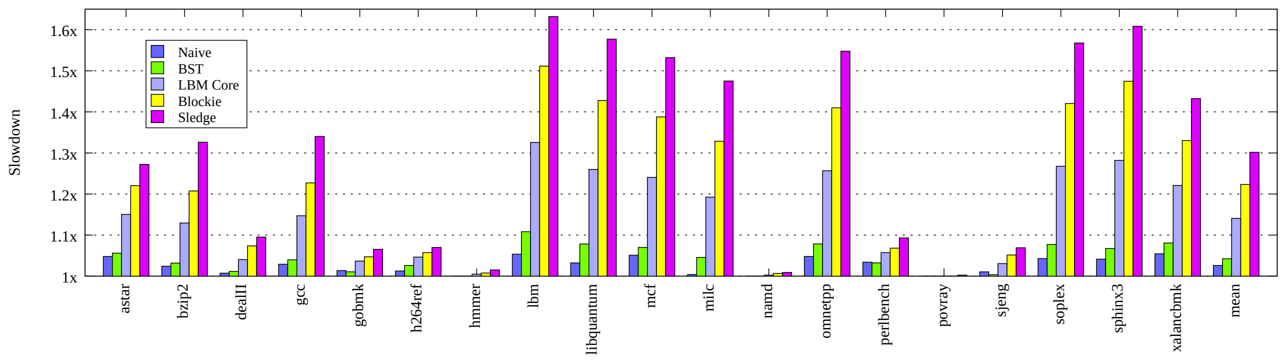

Figure 8: Slowdown caused by contention synthesis on AMD Phenom X4.

`ref` inputs). These benchmarks were compiled with GCC 4.4 on the Linux 2.6.31 kernel. Figure 7 shows the results when performing this co-location on Intel's Core i7 Quad architecture, and Figure 8 shows these results on AMD's Phenom X4 Quad. The bars show the slowdown when co-located with naive random access (naive), binary search tree (BST), the lbm benchmark (LBM Core), the 3D block data movement (Blockie), and our sledgehammer technique (Sledge), in that order. The `lbm` benchmark can be viewed as a baseline to compare the other synthetic engines. We believe `lbm` to be a good point of reference as it is actually a naturally occurring example of a contention application behavior. It is clear from the graphs that the `naive` and `BST` approaches produce the smallest amount of contention. However note that they do an adequate job of indicating the applications that are most sensitive to cross-core interference. The contention produced by these two approaches is low as there is a good bit of computation between single memory accesses. `blockie` and `sledge` touch large amounts of data in a single swoop and with less computation. Note that our Blockie and Sledge techniques are more effective than using the most contentious of the SPEC benchmarks.

Across the two architectures the general trend is similar, although we do see some differences. We see that applications that tend to be sensitive to contention tend to be uniformly so across these two representative architectures. We also see that the varying contention synthesis designs rank similarly on both architectures. This general trend supports our hypothesis that contention is agnostic across this class of commodity multicore architectures.

Although the general trend is the same, there are some clear differences. For example the benchmark most sensitive to cross-core interference on the two architectures differs. On Intel's architecture `mcf` shows the most significant degradation in performance, while on AMD's architecture `lbm` is the clear winner (or loser). These variations are due to the idiosyncrasies of the microarchitectural design.

The key observation is the fact that the effectiveness of the contention synthesis designs are mostly uniform across the different benchmark workloads.

This trend supports our hypothesis that in addition to being generally agnostic across this class of commodity multicore architectures, it is also agnostic across the varying workloads and memory access patterns present in SPEC.

In the following section we selected to use Sledge as our main CSE for our profiling framework as it most vividly illustrates contention across the entire benchmark suite.

5.2 Characterizing Cross Core Interference

To characterize an application's cross core interference sensitivity our profiling framework spawns the *contention synthesis engine* (CSE) on a neighboring core. As the application executes, the profiling runtime directs the CSE to produce short bursts of contentious execution. For every millisecond of execution the profiler will pause the CSE for one millisecond. Slowdowns in the application's instruction retirement rate that result from this bursty execution are monitored using the `instructions_retired` hardware performance counter. A *cross-core interference sensitivity* score is then calculated as the average of these slowdowns.

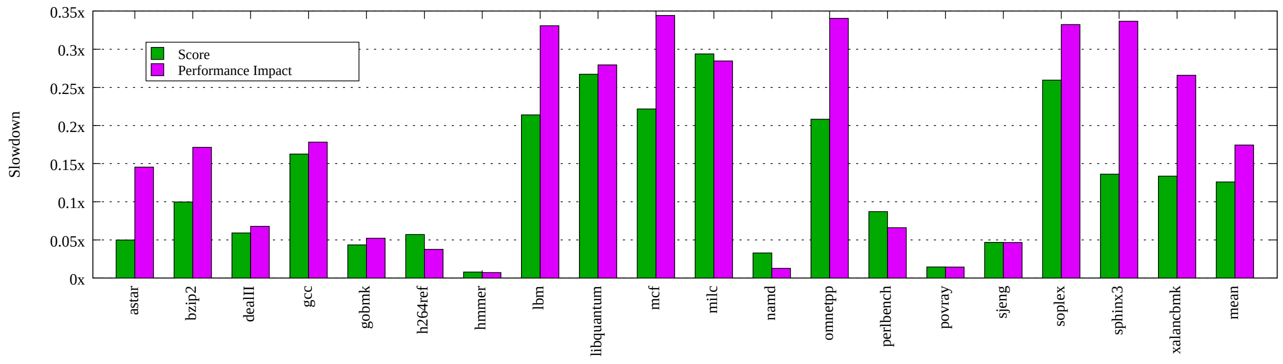

Figure 9: Comparing Characterization Score Trend to Actual Cross-core Interference on Intel Core i7.

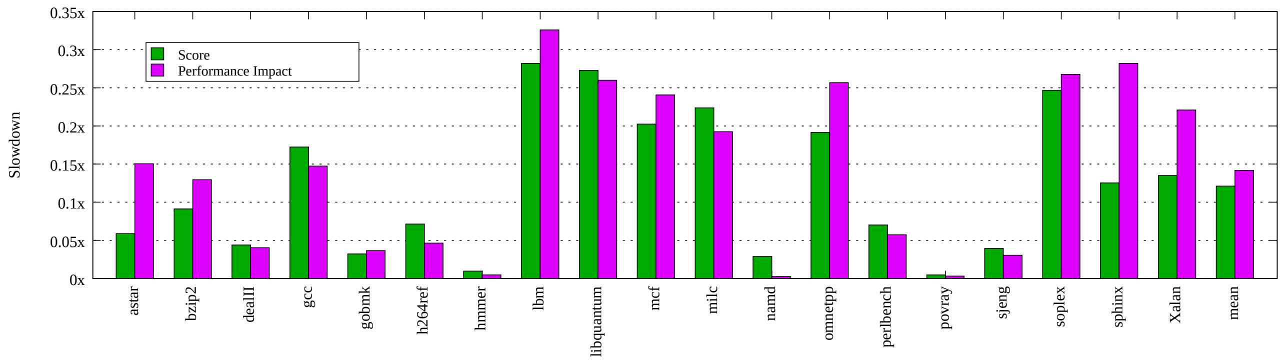

Figure 10: Comparing Characterization Score Trend to Actual Cross-core Interference on AMD Phenom X4.

Figure 9 and 10 show the cross-core interference sensitivity scores calculated using the described method for all C/C++ benchmarks in SPEC2006, compared against the performance degradation when each benchmark is co-running with `lbm`, on both Intel Core i7 and AMD Phenom X4. Our results show that generally, an application's cross-core interference sensitivity score has a strong correlation proportionally with its performance degradation (e.g a lower CIS scores indicate smaller degradations and vice versa). Note that Figures 9 and 10 are intended to demonstrate the trend of how the cross core interference sensitivity score relates to the applications actual degradations due to cross core interference. These figures display a strong trend, indicating that our approach is indeed accurately characterizing cross-core interference sensitivity. However, there are three relative exceptions, `sphinx`, `xalan` and `astar`. These three benchmarks have very clear phases that seem to increase the inaccuracy on its average. One possible way to address this challenge is increasing the periodic probing interval length. Also studying their phase level cross-core interference sensitivity scores would give more insight about their dynamic sensitivity.

We have also experimented with using the change in last level cache misses per cycle to detect contention and measure cross-core interference. Our results show that it is a worse indicator than directly measuring performance degradation using IPCs. Also last level cache misses alone is not always a good indicator either. For example, although an application with a large number of cache misses per cycle because of its heavy cache reliance may in fact be sensitive to cross-core interference, it could also be insensitive if this application already experiences heavy cache misses when running alone. In this latter case cross-core interference would not hurt its already poor cache performance. This often occurs when the working set of the application greatly exceeds the size of the last level cache.

6. RELATED WORK

In this paper we present a profiling and characterization methodology for program sensitivity to cross-core interference on modern CMP architecture. Related to our work is a cache monitoring system for shared caches [25], which proposes novel hardware designs to facilitate better understanding of how applications are interacting and contending when running together. Similar to our work, the system is then used for profiling and program behavior characterization. However, in contrast to our methodology, this work requires hardware extensions and thus is evaluated using simulations. Our methodology and framework is applicable to current commodity multicore architectures. In addition, our framework is not limited to cache contention but any contention in the memory system that would impact performance, and can be produced by our CSE.

In recent years, cache contention has received much research attention. Most works focus on exploring the design space of cache and memory proposing novel hardware solutions or managing policies to alleviate the contention problem. Hardware techniques and related algorithms to enable cache management such as cache partitioning and memory scheduler are proposed [22, 12, 19, 16, 4]. Other hardware solutions to guarantee fairness and QoS include [17, 10, 20].

Related to novel cache designs and architectural support, analytical models to predict impact of cache sharing are also proposed by [2]. In addition to new hardware cache management, approaches to managing shared cache through OS are [21, 5]. Instead of novel hardware or software solution to managing shared caches, our solution focuses on the other side of the problem, namely the application's inherent sensitivity to interference on existing modern microarchitecture.

The idea of profiling applications and learning about their memory-related characteristics from the profile to improve performance and develop more effective compiler optimizations is rather common. Much work has been done for constructing a general framework for memory profiling of applications [18, 9], profiling techniques and methods to use such profiling to improve performance or help develop better compilers and optimizations [9, 24]. Our work is different that we focuses on profiling program's behavior in the presence of cross-core interference.

Contention conscious scheduling schemes that guarantee fairness and increase QoS for co-running applications or multithreaded application have been proposed [13, 6, 1]. Fedorova et al. used cache model prediction to enhance the OS scheduler to provide performance isolation. There are also theoretical studies that investigate approximation algorithms to optimally schedule co-running jobs on CMPs [11, 23].

7. CONCLUSION

In this paper, we present a methodology for profiling and application's sensitivity to cross-core performance interference on current multicore microarchitectures by *synthesizing contention*. Our profiling framework is composed of a lightweight runtime environment on which a host application runs, along with a carefully designed *contention synthesis engine* that executes on a neighboring core. We have explored and evaluated five contention synthesis mechanisms which include the random access of elements in a large array, the random traversal of large linked data structures, a real world fluid dynamics application, data movement in 3D object space commonly found in simulations and scientific computing and finally, we reverse engineered *lbm*, found its most contentious code, and further tweaked it to construct a highly contentious synthesis engine. We have presented the design and implementation of these contention synthesis mechanisms and demonstrated their impact on the SPEC2006 benchmark suite on two real-world multicore architectures. Finally we demonstrate how contention synthesis can be used dynamically, using a bursty execution method, to accurately characterize and applications *cross-core interference sensitivity*.

8. ACKNOWLEDGEMENTS

We would like to acknowledge the important input from Robert Hundt and Neil Vachharajani of Google. Their work and ideas has been instrumental for better understanding the problem of contention, especially as it relates to the data-center and cluster computing domains. We would also like to acknowledge the contribution of Lingjia Tang of the University of Virginia. Her comments and insights have been very helpful to the work and paper. Finally, we would like to thank Google for the funding support for this project.

9. REFERENCES

[1] M. Banikazemi, D. Poff, and B. Abali. Pam: a novel performance/power aware meta-scheduler for multi-core systems. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[2] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[4] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 242–252, New York, NY, USA, 2007. ACM.

[5] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, Washington, DC, USA, 2006. IEEE Computer Society.

[6] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 25–38, Washington, DC, USA, 2007. IEEE Computer Society.

[7] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[8] Intel Corporation. *IA-32 Application Developer's Architecture Guide*. Intel Corporation, Santa Clara, CA, USA, 2009.

[9] M. Itzkowitz, B. J. N. Wylie, C. Aoki, and N. Kosche. Memory profiling using hardware counters. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 17, Washington, DC, USA, 2003. IEEE Computer Society.

[10] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 25–36, New York, NY, USA, 2007. ACM.

[11] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 220–229, New York, NY, USA, 2008. ACM.

[12] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor

architecture. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

[13] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, 2008.

[14] S. Lohr. Demand for data puts engineers in spotlight. *The New York Times*, 2008. Published June 17th.

[15] J. Mars and R. Hundt. Scenario based optimization: A framework for statically enabling online optimizations. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 169–179, Washington, DC, USA, 2009. IEEE Computer Society.

[16] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 208–222, Washington, DC, USA, 2006. IEEE Computer Society.

[17] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 57–68, New York, NY, USA, 2007. ACM.

[18] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[19] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[20] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 2–12, New York, NY, USA, 2006. ACM.

[21] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 258–269, Washington, DC, USA, 2008. IEEE Computer Society.

[22] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 117, Washington, DC, USA, 2002. IEEE Computer Society.

[23] K. Tian, Y. Jiang, and X. Shen. A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. In *CF '09: Proceedings of the 6th ACM conference on Computing frontiers*, pages 41–50, New York, NY, USA, 2009. ACM.

[24] Q. Wu, A. Pyatakov, A. Spiridonov, E. Raman, D. W. Clark, and D. I. August. Exposing memory access regularities using object-relative memory profiling. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 315, Washington, DC, USA, 2004. IEEE Computer Society.

[25] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. Cachescouts: Fine-grain monitoring of shared caches in cmp platforms. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 339–352, Washington, DC, USA, 2007. IEEE Computer Society.