Harmony: Collection and Analysis of Parallel Block Vectors

Melanie Kambadur, Kui Tang, and Martha A. Kim
Columbia University, New York, NY
{melanie,martha}@cs.columbia.edu, kt2384@columbia.edu

Abstract

Efficient execution of well-parallelized applications is central to performance in the multicore era. Program analysis tools support the hardware and software sides of this effort by exposing relevant features of multithreaded applications. This paper describes parallel block vectors, which uncover previously unseen characteristics of parallel programs. Parallel block vectors provide block execution profiles per concurrency phase (e.g., the block execution profile of all serial regions of a program). This information provides a direct and fine-grained mapping between an application's runtime parallel phases and the static code that makes up those phases. This paper also demonstrates how to collect parallel block vectors with minimal application perturbation using Harmony. Harmony is an instrumentation pass for the LLVM compiler that introduces just 16-21% overhead on average across eight Parsec benchmarks.

We apply parallel block vectors to uncover several novel insights about parallel applications with direct consequences for architectural design. First, that the serial and parallel phases of execution used in Amdahl's Law are often composed of many of the same basic blocks. Second, that program features, such as instruction mix, vary based on the degree of parallelism, with serial phases in particular displaying different instruction mixes from the program as a whole. Third, that dynamic execution frequencies do not necessarily correlate with a block's parallelism.

1 Introduction

As multi-cores have come to dominate programmable architectures from mobile to the datacenter, efficiency in parallel programming has seen significant attention from both research and industry. Parallel profilers and measurement tools have helped application parallelization, often by exposing hard to identify parallel performance issues. Intel's VTune [13] and the gprof-based Kremlin [6] are examples of such tools. While these tools are certainly useful to software engineers, they don't capture the whole picture of a parallel program's execution.

This paper introduces parallel block vectors, profiles which establish a mapping between static basic blocks in a multithreaded application and the degrees of parallelism exhibited by the application each time a basic block executes. These profiles enable the discovery of two previously unseen characteristics of parallel programs: they tease apart serial and parallel portions of a program for individual analysis, and they track the changes in parallelism of fine-grained code regions. A parallel block vector consists of an array of counters where each counter $counter_{b,t}$ indicates how many times basic block b was executed when the application had t threads running. From this profile it is easy to find blocks that executed at a particular thread count t (e.g., all b such that $counter_{b,t} > 0$), or the thread counts each time a particular block b was executed ($counter_{b,t}$ for all values of t).

This paper shows that with careful engineering effort, parallel block vectors are neither complex nor expensive to gather even at such fine granularity. We demonstrate Harmony, an LLVM compiler pass that instruments a multithreaded application to gather parallel block vectors. For eight Parsec benchmarks, instrumentation using Harmony incurs an average of 16% application slowdown and has minimal resource overhead as measured by register spills and cache miss rates.

The new parallel program characteristics uncovered by parallel block vectors can be used to improve multithreaded execution in a variety of ways. We demonstrate several discoveries made via analysis of parallel block vectors that are relevant to microarchitectural design. For example in Section 5.2, we use parallel block vectors to separate the parallel and serial code portions of several applications, discovering that the instruction mixes for these subsets of code differ, often significantly, from the overall program instruction mix. In the context of heterogeneous processors, such as those analytically motivated by Marty and Hill [11], this information can be applied to tailor heterogeneous cores to better suit their anticipated parallel and serial workloads. Parallel block vectors can also be applied to a variety of other multi–threaded performance issues. We suggest applications in software engineering, application scheduling, and compilers research in Section 6.

In summary, this paper makes the following three contributions:

- Defines parallel block vectors, a novel way of measuring parallel program performance that can reveal previously unseen multithreaded program features.
- Describes Harmony, a tool that allows fast (only 16% slower than runtime) and accurate collection of parallel block vectors via compiler inserted instrumentation and dynamic profiling.
- Demonstrates three applications of parallel block vectors, discovering that: (1) In many cases the black and white scenario of Amdahl's Law, in which code is either purely serial or purely parallel, does come to pass, with blocks displaying strong affinities for either serial or parallel execution. However, there are also exceptions in which substantial numbers of basic blocks run both serially and in parallel across different executions. (2) Program features, such as instruction mix and basic block size, vary across blocks that can be categorized into different degrees of parallelism. Notably, features of identifiably serial blocks often differ significantly from whole program features. (3) The frequency of execution of a block does not necessarily correlate to parallelism or serialism. This suggests that when "hotspot" analysis is used in the context of processor design, architects should consider parallelism as a factor in their analysis.

The remainder of this paper discusses these contributions in further detail. Section 2 defines parallel block vectors and shows a sample parallel block vector for a simple matrix multiply application. Section 3 describes Harmony, a static instrumentation tool to collect the profiles, and Section 4 quantifies the low overheads and minimal application perturbation due to profiling. Section 5 uses analysis of parallel block vectors to make our three architectural discoveries, and Section 6 discusses ideas for future uses of parallel block vectors.

2 Parallel Block Vector Profiles

Many profiling tools collect runtime statistics from the perspective of processes or threads [13, 12, 32, 15], reporting the number of threads running for the duration of a process or the breakdown of serial and parallel execution time. Parallel block vectors report on a program's parallel behavior from the perspective of a basic block. A parallel block vector consists of one histogram for each basic block, indicating the degree of parallelism exhibited by the application each time the block was executed.

Figure 1 shows a parallel block vector profile for a simple, unoptimized matrix multiplication program. The profile shown in the table is a matrix with one row for each

```
#define NDIM 1000
double a[NDIM][NDIM];
double b[NDIM][NDIM];
double c[NDIM][NDIM];

void worker(int me,int p,int n) {
  int    i,j,k;
  double sum;
  i = me;
  while (i < n) {
    for (j = 0; j < n; j++) {
      sum = 0.0;
      for (k = 0; k < n; k++)
        sum = sum+a[i][k]*b[k][j];
      c[i][j] = sum;
    }
    i += p;
  }
}
```

```
int main(int argc, char *argv[]) {
  // Variable declaration and
  // initialization ommitted
  n = // number of threads, here 4
  threads = (pthread_t*)
    malloc(n*sizeof(pthread_t));
  pthread_attr_init(&pthread_custom_attr);
  for (i = 0; i < n; i++)
    pthread_create(&threads[i],
                   &pthread_custom_attr,
                   worker, ...);
  for (i = 0; i < n; i++)
    pthread_join(threads[i], NULL);
  free(arg);
  return 0;
}
```

	Nominal Thread Count				
	1	**2**	**3**	**4**	**5**
main:9	0	0	0	0	1
worker:7	0	14	14	16	956
worker:6	606	146K	12K	16K	955K
worker:5	607K	14.6M	12.8M	16.1M	955M
worker:4	607	146K	12K	16K	955K
worker:3	1	14	14	16	955
worker:2	0	1	1	1	1
worker:8	0	1	1	1	1
worker:1	1	1	1	1	0
worker:0	1	1	1	1	0
main:7	3	0	0	1	0
main:6	1M	0	0	0	0
main:5	1K	0	0	0	0
main:4	1K	0	0	0	0
main:3	1	0	0	0	0
main:2	1	0	0	0	0
main:1	1	0	0	0	0
main:0	1	0	0	0	0

Figure 1: **Parallel block vector for matrix multiplication.** For each basic block in an application, top, the profile, bottom, indicates the block execution frequency at each possible thread count (i.e., degree of parallelism).

static basic block and one column for each possible degree of concurrency. In this example, the program created four threads, which in addition to the initial thread, makes at most five concurrent threads. Each cell in the profile gives the number of dynamic executions of the given block at the given degree of parallelism. To help survey large applications, we also use the heatmap visual representation shown by the shading in Figure 1.

Parallel block vectors create two new opportunities for better understanding parallel programs. First, they allow identification of specific basic blocks that run at a particular thread count. For example, examining the first column in Figure 1 reveals that fourteen blocks make up the serial phases of matrix multiplication's execution, with main:6 and worker:5 dominating the dynamic mix. Second, a user can monitor regions of interest in a program to see the phase or phases in which the code executed. For example, blocks worker:4-6 in Figure 1 correspond to the inner multiplication loop, a critical region in terms of performance. As one would hope, the profile reveals that this code is largely executed at high thread counts.

There are multiple ways to count threads when determin-

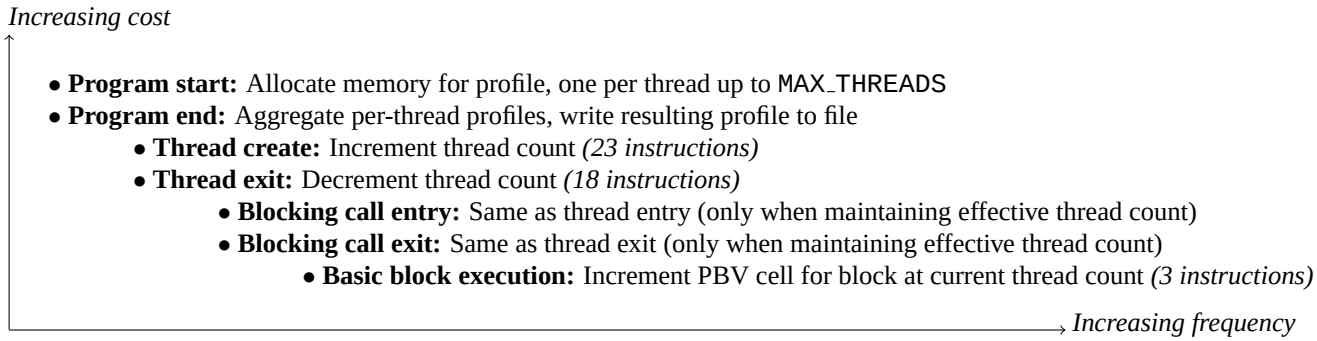

Figure 2: **Harmony instrumentation points.** Profiler action is taken upon various runtime events. Careful engineering offloads expensive work to the least frequent events, in particular program start and finish which do not overlap with the execution of the program itself. This results in minimal profiling work at the most frequent events (i.e., basic block executions), reducing the profiling overhead and minimizing perturbation.

ing the parallel phase of an application. *Nominal thread count* includes all created threads regardless of whether they are running or blocked. *Effective thread count* excludes blocked threads and counts only runnable threads. *Running thread count* includes only the runnable threads that have actually been granted access by the operating system to a processor. We collect profiles for nominal and effective thread counts, but do not count running threads for two reasons. First, running thread count is strongly dependent on the availability of hardware resources and the behavior of the scheduler, thus revealing more about those two aspects of the system than the application. Second, counting running threads requires polling the OS, which is likely to substantially slow and perturb the execution of the program under measurement.

3 Harmony: Efficient Collection of Parallel Block Vectors

We now describe Harmony, an instrumentation pass for LLVM [19] to generate parallel block vectors. We selected compile-time instrumentation for Harmony for three reasons. First, parallel block vectors require dynamic information such as basic block execution frequency, thread count, and timing information which is not available via static analysis. Second, unlike dynamic instrumentation frameworks such as Pin, compile-time instrumentation adds no additional runtime overhead beyond the instrumentation code itself. It is particularly important to keep overheads low when profiling parallel applications as shifts in the relative timing of events can perturb the behavior of the program. Finally, with compile-time instrumentation, portability comes for free, making it trivial to collect profiles on any architecture or language supported by the compiler.

This section describes the architecture of Harmony and discusses the efforts undertaken to minimize profile collection overhead thereby maintaining profile accuracy. The pass is intended to be the last pass executed, after the program has been fully optimized and the final program control-flow graph (CFG) has been set. Harmony is available as an open-source tool at `http://arcade.cs.columbia.edu/harmony`.

3.1 Injecting Instrumentation

To collect parallel block vectors, Harmony must take action at several program events, as summarized in Figure 2. At program start the profiler must allocate and initialize a profile, and at program finish the profile must be written to a file. At thread creation and exit, Harmony must inject code to increment and decrement the nominal thread count. When tracking effective thread count, the counter must also be decremented upon entry and incremented upon exit from any blocking call, such as a lock acquire. Lastly, each basic block execution must be accompanied by an increment of the appropriate entry in the profile matrix.

Harmony injects instrumentation in two different ways. For tracking basic block executions, Harmony adds instructions directly into the body of a basic block, as illustrated in Figure 3. The same goes for program entry and exit, where Harmony inserts calls to profile initialization and cleanup routines (not shown). For the remaining events, Harmony interposes on relevant thread library calls as illustrated in Figure 4. At present, the tool supports only Pthreads library calls, and requires only that programs include `harmony.h` in place of `pthreads.h`.

3.2 Strategies for Minimizing Perturbation

Adding instrumentation to a parallel program risks perturbing program behavior, potentially compromising the accuracy of the profile. While some perturbation is unavoidable, we found that careful engineering significantly reduces the overhead of profile collection.

As basic block executions are by far the most frequent event the profiler instruments, we focused our op-

```
void sample(uint32_t bb_id) {
  bucket_t *b = *ptr_specific_col +
            bb_id*PROF_BUCKET_SIZE;
  (*b)++;
}
```

```
# load the pointer to pointer to my column
movl %gs:ptr_specific_col@NTPOFF, %edx
# load the pointer to my column
movl (%edx), %edx
# increment counter for BBL1 at specific_col+4
incl 4(%edx)
```

Figure 3: **Direct instrumentation example.** Each basic block is augmented to record its execution at the current degree of parallelism. The additional three instruction use only one register and do not induce any register spills.

```
// intercept potentially blocking call
#define pthread_mutex_lock(a...) \
   BLOCKING_CALL(pthread_mutex_lock(a))

// effective thread count drops on entry
// and rises on upon completing
#define BLOCKING_CALL(exp) ({ \
   int rv; \
  __sync_sub_and_fetch(&(effectiveThreadCount), 1); \
   rv = exp; \
  __sync_add_and_fetch(&(effectiveThreadCount), 1); \
   rv; })
```

Figure 4: **Thread library wrapper example.** Here the instrumentation decrements and increments the effective thread count upon upon entry to and exit from of a blocking call respectively.

timization efforts there. Each time a basic block executes, the instrumentation must read the current thread count and use that value along with the basic block ID to index the profile matrix and increment one counter (i.e., `profile[currentThreadCount][bbid]++`). Harmony takes the following steps to streamline this computation:

- Because `bbid` will be changing much more frequently than `currentThreadCount`, the profile matrix is laid out in a cache-friendly, column-major fashion that places profile entries for different basic blocks at the same degree of parallelism at adjacent addresses in memory.
- Significant portions of the address calculation are factored out of the basic blocks themselves. Specifically, the column address offset for the current thread count need only be re-calculated each time the thread count changes and not for each basic block execution. All that remains of the address calculation for each basic block is to compute the offset within the profile column.
- Finally, because the target programs are parallel, multiple threads will be updating the profile concurrently. Rather than guarding each counter in the profile matrix with a lock, which would introduce substantial synchronization overhead, we allocate a private profile matrix for each thread, and aggregate the per-thread profiles only after the program has finished executing.

Collectively, these optimizations result in the small per-basic block overhead of the three instructions shown in Figure 3.

3.3 Mapping Profiles Back to Application Code

To ensure that profiles can be mapped back to the original application code, Harmony annotates both the profiles and the LLVM assembly file with unique basic block IDs. In post-processing these two files can be cross-referenced for further analysis as in our instruction mix case study (Section 5.2). Though we do not implement it for these studies, this labeling scheme could be coupled with debug symbols to link the profile all the way back to source code.

3.4 Limitations

At present Harmony is usable only on Pthreads applications that LLVM can compile. The tool could be extended to support other parallelization libraries (e.g., OpenMP), and the general architecture could be readily ported to other compilers.

4 Runtime Impact of Harmony

Dynamic analysis risks altering the timing, and with it the behavior, of a parallel program in a way that may compromise the accuracy of the gathered information. For example, slowing critical sections will increase lock contention, and, conversely, slowing non-critical sections will reduce lock contention. It is thus important to carefully examine profiling's impact on the original program.

In his 1991 paper, *Event-based Performance Perturbation: A Case Study*, Allen Maloney [20] listed the three primary sources of program perturbation: execution of additional instructions and their resulting execution slowdown, changes in memory references patterns, and register pressure. As outlined in Section 3.2, Harmony takes a number of steps to minimize the impact on program behavior. In this section we evaluate the profiler's impact on each of these three metrics.

4.1 Execution Time Overhead

First, we compare the execution time of applications compiled with and without Harmony's instrumentation. In both cases the `-O3` flag is set to turn on maximal compiler optimizations. The machine used for all experiments described in this paper has 4 2.0 GHz cores, 3.3GB of RAM,

and is running Linux Ubuntu version 8.04. We use Harmony to collect parallel block vectors for eight applications from Parsec [4], a suite of non-HPC multithreaded benchmarks.

All runtimes are the average of 20 program runs. For each application the profile collection times were normalized to an uninstrumented baseline. Figure 5 plots the profile collection overheads. Nominal thread count profiling added 16% on average while effective thread count profiling added slightly more overhead at 21%. The additional overhead is expected due to the additional thread counter activity. As Figure 5 indicates, 3% of these totals are attributable to time spent writing the profile to a file after the program has finished. Thus, the effective overheads *during program execution* are 13% and 18%, respectively.

Relative to similar tools, these overheads are modest. For example, ThreadScope, a tool for tracing runtime parallel events using the Haskell GHC compiler [17], incurs 10%-25% overhead. Quartz, a gprof like tool that uses sampling to monitor threads, increases program run times by 70% [1]. The popular (but heavier-weight) runtime binary instrumentation platform, Pin, incurs a $100 - 400\%$ increase in execution time for basic block counting alone [3].

It is interesting to note that two applications, `dedup` and `fluidanimate`, spend significantly more time maintaining an effective thread count than maintaining a nominal thread count. This differential in activity between the two counters becomes significant when we compare the resulting profiles later in Section 5.1.

4.2 Storage Resource Contention

The last two sources of perturbation in Maloney's list address increased resource pressure caused by instrumentation. For Harmony we see a slight — 7.5% on average — increase in register spills. However, for these applications, the additional spills were confined to the profile setup and cleanup activities which occur prior to and after the execution of the program itself. Most importantly, the instrumentation code in each basic block did *not* induce spills.

As measured by Cachegrind [34], the instrumentation introduces negligible cache perturbation. In the L1 instruction and data caches the miss rate increased by at most 0.06% and 0.2% respectively. There was no measurable impact on the hierarchy beyond the L1 structures (i.e., L2 miss rates were unchanged).

5 Applications of Parallel Block Vectors

We now carry out three novel analyses of our benchmarks, each enabled by parallel block vectors. Figure 6 shows visual representations of the profile of each of the eight Parsec benchmarks. Recall from Figure 1 that each row corresponds to a static basic block and each column to a nominal thread count. For space reasons we show the full heatmaps only for nominal thread counts, though in the following sections we will analyze both nominal and effective thread count profiles.

From these profiles, we see that in several applications (namely `bodytrack`, `dedup`, `facesim`, `fluidanimate`, and `streamcluster`) basic blocks display a strong affinity for either serial or parallel phases. The remaining applications (`blackscholes`, `swaptions` and `x264`) by contrast have significant portions which execute at mixed thread counts, during both serial and parallel phases.

It is well known that the serial portions of an application limit parallel speedups [11], but what exactly do those serial portions look like? Are they amenable to acceleration? We will explore these questions in the following sections, before closing with a discussion of other applications of parallel block vectors.

5.1 Serial and Parallel Application Partitions

Knowing how much of a program runs in parallel and how much runs serially is useful for many purposes. Tools such as Intel's VTune Amplifier XE [13] identify the serial fraction of an application's runtime so that software engineers can improve the parallelization of their programs. Such metrics are also useful when estimating the scalability of a particular parallelization according to Amdahl's Law [11].

Parallel block vectors make it possible not only to quantify the serial portion of a program, but to map that region back to the specific basic blocks that comprise it. To get this information we classify each basic block into one of three categories: *serial* (i.e., never executed with a thread count greater than one), *parallel* (i.e., always executed with a thread count greater than one) or *mixed* (i.e., sometimes executed in serial regions, other times in parallel regions).

From an architect's perspective, the pure serial blocks make natural targets for specialized serial processors (further discussed in Section 5.2) or accelerators (Section 5.3). The mixed blocks, which run both in parallel and serially, are likely of interest to all system designers. They might represent areas in the application where there were communication overheads or other forms of architectural resource contention. Identifying the mixed blocks allows their execution to be improved with better scheduling algorithms, additional hardware resources, or code transformations.

Figure 7 shows the breakdown of static and dynamic basic blocks by class (serial, mixed, or parallel). We observe that significant portions of several applications are neither purely serial nor purely parallel, but rather belong to both regions (the mixed class). This is true of both nominal and effective thread counts. The trends are similar but even more pronounced when counting dynamic basic block executions. This means that when we talk about Amdahl's law and serial and parallel phases of a program, those phases

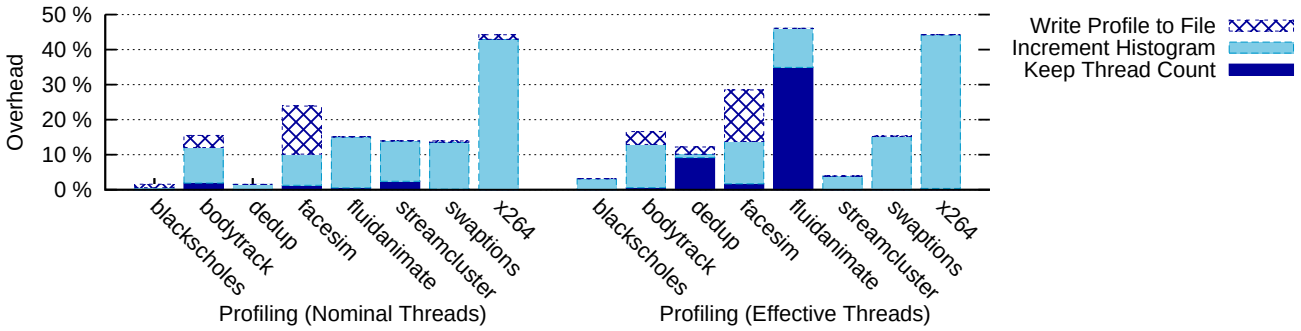

Figure 5: **Low overhead of instrumentation.** Program slowdown due to profile collection ranges from 2% to 44% with an average overhead of 18%.

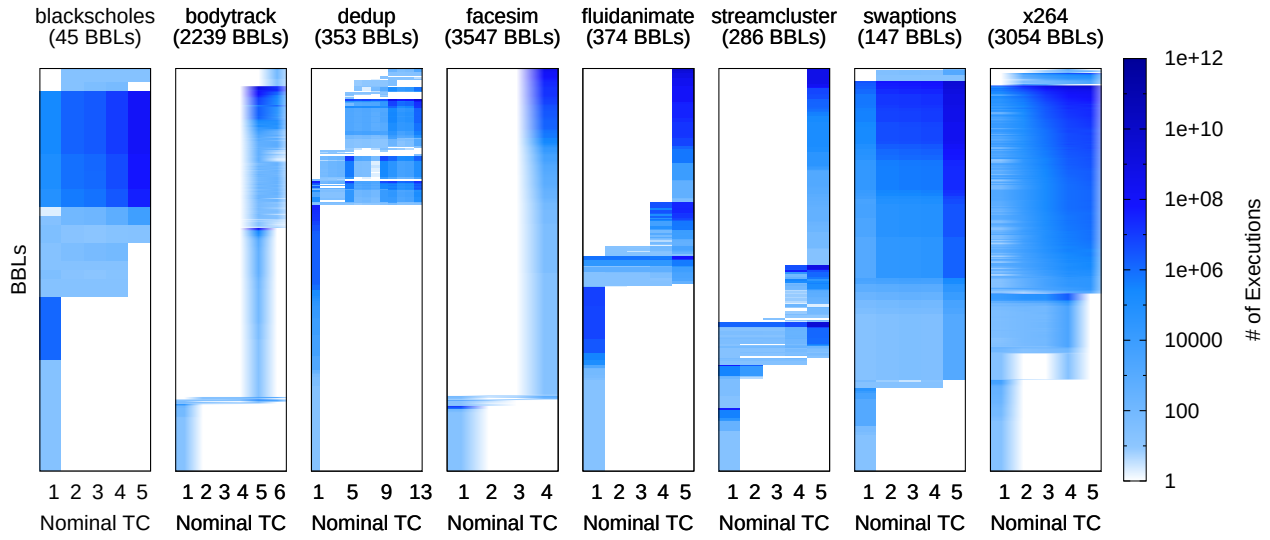

Figure 6: **Parallel block vectors for Parsec.** These heatmaps are a visualization of the profiles produced by Harmony. For the given application, they show the number of times (shading) each static block (row) was executed at each degree of parallelism (column).

often *do not* correspond to different portions of the application. One hypothesis is that such blocks are the result of library code which is called both from the serial and parallel phases.

Returning to the serial/mixed/parallel classifications, we can also clearly tell which applications are the most parallel. For example, from the static nominal view, `bodytrack` and `facesim` seem to be equally parallel. However, from the dynamic effective profiles, we see that `bodytrack` has more blocks actually running in parallel, whereas `facesim` apparently suffered from blocking threads and its parallel blocks were less frequently executed than its serial blocks. In the following section, we will look in more depth at the content of blocks in each of these classes.

5.2 Program Features by Degree of Parallelism

Recent interest in heterogeneous multicore architectures spans not only the architecture community but operating systems, high-performance computing, programming languages, and others [14, 37, 18, 23, 2, 8, 27]. The principle idea behind heterogeneous processing is specialization: different cores on a heterogeneous machine can address the varied compute needs of modern workloads while maximizing hardware performance and efficiency. For example, when portions of a program cannot be adequately parallelized, an aggressive, out of order, no holds barred processor might be employed to reduce execution time.

If heterogeneous cores are meant to address the specialized needs of certain portions of the application, it is naturally important to understand what these processors should

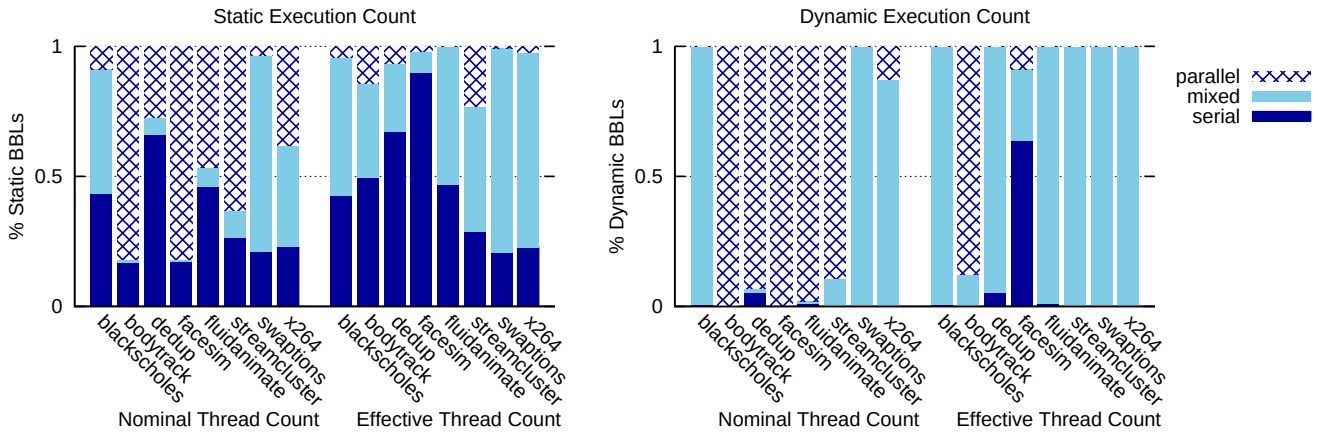

Figure 7: **Classifying basic blocks by parallelism.** These graphs show the percentage of blocks which execute only serially (serial), blocks which execute both serially and in parallel (mixed), and blocks which only execute in parallel (parallel) for each application, for both nominal and effective thread counting, and for both static and dynamic block executions.

be specialized to. Parallel block vectors can assist by distinguishing features of parallel and serial phases. For this analysis, we will continue to use the *always serial*, *always parallel*, or *mixed* classification introduced in Section 5.1 in which every block belongs to exactly one class.

Figure 8 compares the dynamic instruction mixes of each of these three categories, as well as for the program as a whole. All of the X86 opcodes that occurred in the application were classified into one of eight categories: loads and stores, loads of effective addresses, integer arithmetic, floating point arithmetic, comparisons, conditional control transfers, unconditional control transfers, and synchronization.

We observe that for most applications, serial basic blocks display significantly different instruction mixes from the overall program. This indicates an opportunity for architects to exploit, when designing the microarchitecture of aggressive cores for heterogeneous CMPs. Consider the `blackscholes` application. Across the whole program, floating point operations account for more than 20% of the dynamic instructions. If this were the only instruction mix considered, as is currently the case, then the aggressive processor for serial regions might waste space and expense unnecessarily on floating point units, when we can see from the graph that the serial blocks actually require fewer floating point operations than the program as a whole. Instead, the serial phases of `blackscholes` have a higher concentration of control and integer arithmetic, suggesting that resources would be better spent on the branch predictor, for example.

The data in Figure 8, shows such a pattern in each of the benchmarks. In every case, either the serial or parallel portions (and sometimes both) have substantially different instruction mixes than the application as a whole. However, across these applications, there does not appear to be a consistent pattern of *how* the instruction mixes change. For example, the serial portions in `blackscholes` had reduced need for floating point units, while the serial portions of `x264` show increased rates of unconditional control transfers. It is not immediately obvious how hardware can or should exploit such patterns. We believe that this direction merits further investigation, starting with a more comprehensive review of the application space.

Just as opcodes vary, the state upon which the serial and parallel portions of a program operate varies relative to the overall program. Figure 9 shows the memory interactions of the three parallelism classes. As with opcodes, the component parts of the application show different mixes than the application as a whole.

5.3 Hotspot Analysis Using Parallel Block Vectors

The previous section suggests an approach for using parallel block vectors to determine the applicability of a specialized processor to particular code regions. An extreme form of specialized processor, accelerators have shown great promise in reducing power, saving space in embedded systems, and improving performance for target programs. The following case study explores how Harmony can help architects quantify the potential performance gains of their accelerator designs, in particular how parallel block vectors can enhance hotspot analysis for parallel applications.

Figure 10 plots *average degree of parallelism* against dynamic basic block executions for block in each application. The average degree of parallelism of a block is simply the average thread count for each block weighted by the block's execution frequency at each count. The scatter

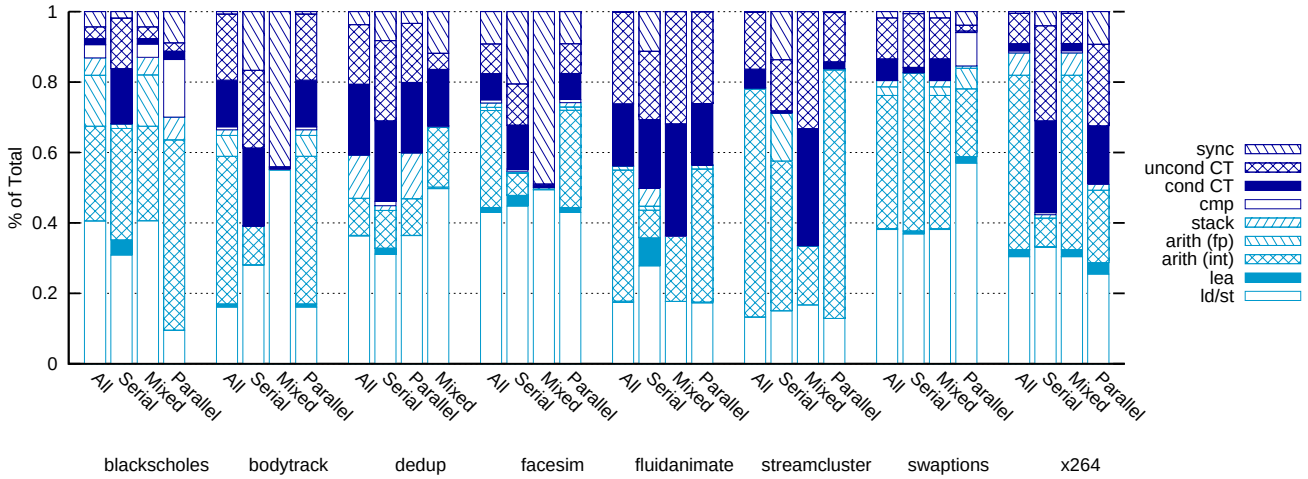

Figure 8: **Opcode mix by class.** Instruction mixes for the entire program compared with the mixes for each basic block class (serial, parallel, and mixed). In all applications, the instruction mixes for both purely serial and purely parallel blocks differ significantly from whole program mixes.

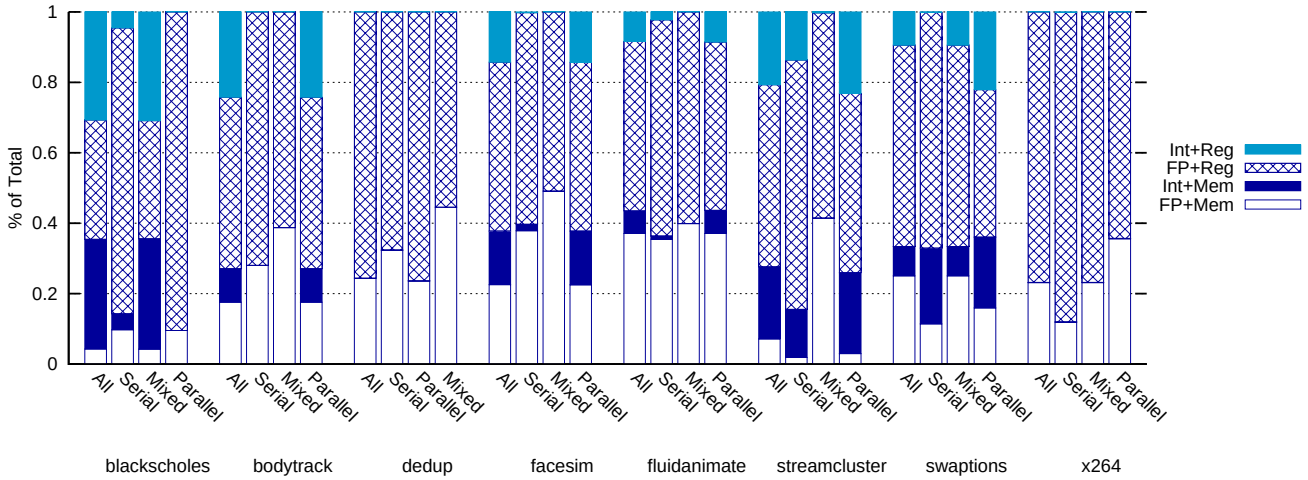

Figure 9: **Memory interaction by class.** The proportion of memory operations for serial and parallel basic blocks differ from the proportion in the program as a whole.

plots reveal that the hottest blocks are not always the most parallel ones. In `streamcluster` for instance, many of the hottest blocks have an average degree of parallelism of one. Generally, the hottest blocks seem to be split between blocks which execute exclusively serially and blocks which execute at or near the maximum degree of parallelism. This data indicates that not only are there hotspots, possibly amenable to acceleration, but that one should not assume anything about whether the hotspots belong to parallel or serial phases. Some code simply cannot be parallelized. As multicore architectures scale to larger core counts, these serial portions of runtime dominate total execution times. The acceleration of serial sections then becomes critically important. So, as a special case of hotspot analysis, we look more closely at the serial blocks, and ask the question, *are serial code segments amenable to targeted accelerator optimizations?*

Taking the serial basic blocks identified in Section 5.1 (see Figure 7), we attribute dynamic serial execution frequencies to different percentages of the serial blocks. Figure 11 (left) shows that for six of the eight applications, 75% of the serial execution is attributable to less than 10% of the basic blocks. This data corroborates what other projects [35] have seen, that accelerators can effectively accelerate the serial parts of a parallel application.

Processor designers might also be interested in how amenable parallel blocks are to targeted hardware optimizations. Figure 11 (right) shows the execution coverage of parallel phases by purely parallel basic blocks. With the exception of `blackscholes` and `swaptions`, ap-

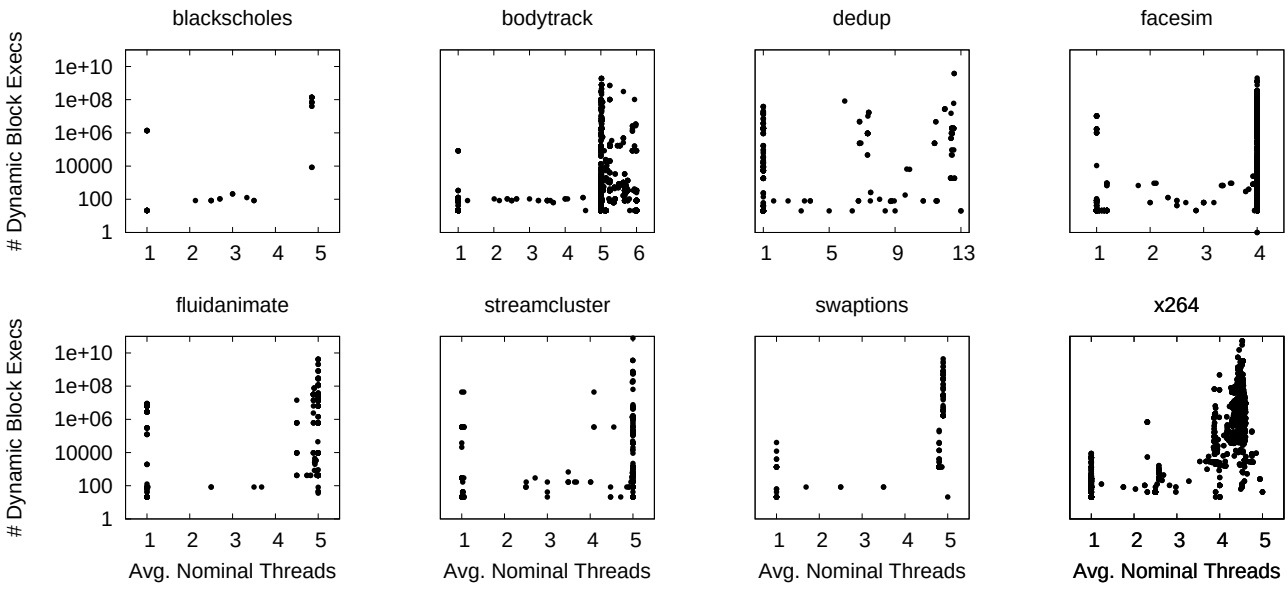

Figure 10: **Hottest blocks are not always the most parallel blocks.** Each static basic block's weighted average nominal thread count was calculated and then plotted against its total number of dynamic executions. The graphs show that the hottest blocks are primarily split between those that execute only serial and those that execute near the max degree of parallelism.

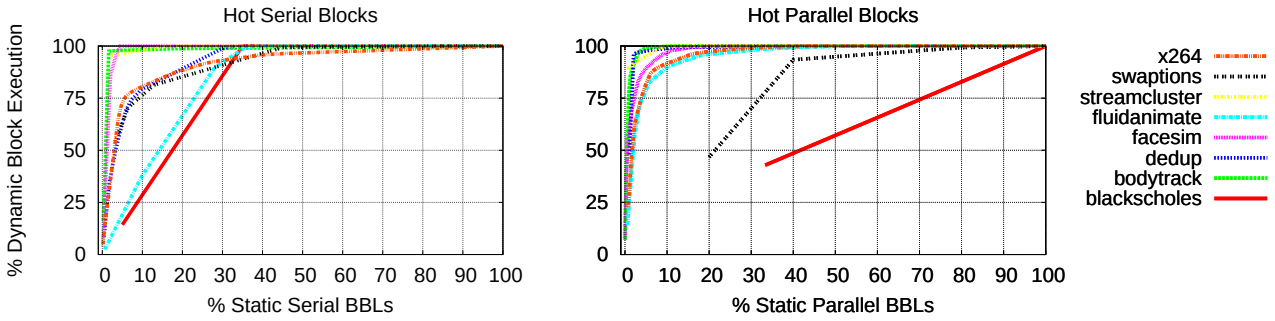

Figure 11: **Few basic blocks represent large portions of serial and parallel runtime.** For basic blocks that were determined by parallel block vectors to always execute serially (left) or in parallel (right), percentages of runtime execution are attributed to static basic blocks. For most applications, a small number of blocks represents a large fraction of the total runtime.

proximately 5% or fewer of parallel blocks are responsible for 75% of dynamic parallel blocks. The reason that `blackscholes` and `swaptions` do not show a steep hotspot curve is that they had very few parallel blocks to begin with; three and five, respectively. As with serial code, we find that parallel parts of the applications exhibit pronounced hotspots.

In the above experiments, we examine hotspots in terms of basic blocks, but only because this was the most natural first choice given that it matched our profile granularity. We note that similar experiments can easily be run at hot function or hot instruction granularity if we statically analyze the basic blocks and source program after running Harmony. It would also be possible, with some additional effort, to map hot call graph or dataflow paths to parallelism.

6 Other Applications and Future Work

Parallel block vectors are flexible, light-weight and applicable for a range of multi-disciplinary uses. We have already discussed several analyses directly relating to hardware design. Here we discuss other applications spanning software engineering, operating systems, compilers, and machine learning. These ideas are meant to demonstrate the diverse utilities of parallel block vectors, and to inspire readers to find further uses.

6.1 Applications in Software Engineering

Writing parallel software is a challenging task. One particular challenge lies in verifying that applications consistently run as the developer expects. Harmony could assist this verification process by checking that particular parts of the program run at the degree of parallelism intended by the developer. For example, a language could introduce assertions to declare that a specific code region should never run when the thread count is greater than one. This might be a critical section, or it might be any other code region that a developer expects to execute serially. Then, Harmony could be modified to insert runtime checks and flag them for programmer inspection.

Another concurrency check that Harmony could assist with is the identification of code regions with *anomalous parallelism*. If a certain code region, say a function, is found to run serially 99% of the time and in parallel 1% of the time, this anomaly might signify a concurrency bug, or at least a potential mismatch in programmer intent and runtime behavior and could also be flagged for programmer review.

6.2 Applications in Operating Systems Research

As we observed in Section 5.1, many applications have a significant fraction of mixed parallelism blocks. These blocks might be indicative of poor operating system scheduling. Further examination of such mixed blocks could lead to improvements in scheduling policy.

6.3 Applications in Compilers Research

If compilers are knowledgeable about the degree of parallelism at which a basic block might run at, optimization selection could factor in this information. Multi-threaded programs might initially be optimized as if they were to be executed in serial, then run with Harmony profiling. The parallel block vectors produced could be used by a compiler to apply different optimization strategies to parallel and serial code. For example, if a heterogeneous CMP has in-order parallel cores, the compiler might expend more effort on instruction scheduling.

Profile driven re-compilation could also be employed when targeting code to specialized processors in a heterogeneous architecture. An initial run of an application with Harmony profiling followed by instruction analysis could determine the best processing unit on which to run a particular code region. Re-compilation could then prepare the application to run on specialized cores, potentially with different instruction set architectures (ISAs).

Mapping measured parallelism to basic blocks might also help a compiler improve program parallelism. Parallel classifications like always serial, always parallel, and mixed could be mapped to control flow graphs. The attribution of parallelism to CFGs might highlight certain graph patterns where opportunities for further parallelization exist.

6.4 Machine Learning

We chose the always serial, always parallel, mixed classifications because they are appropriate to the microarchitectural design case studies presented. However, blocks could be classified in a multitude of ways. For example, we identified blocks which always ran in parallel, but did not distinguish blocks which were highly parallel from blocks which were only somewhat parallel. That is, we did not separate blocks which ran concurrently with five other threads active from those that ran with one other thread. Different classifications might be useful depending on the profiling goal and the type of application being measured. Unsupervised learning could determine useful parallel classifications, leading to more interesting analyses and to further insights for a variety multi-threaded applications.

7 Related Work

To the best of our knowledge, Harmony is the first tool that dynamically records parallelism and maps it back to basic blocks in the application. However, a number of other tools dynamically measure program parallelism and profile thread activity, use the LLVM compiler, or collect basic block-granularity performance data.

Parallelism Analysis Tools. Kremlin by Garcia et al. [6] reboots the classic gprof [7] for the multicore era using hierarchical critical path analysis to help users identify application hotspots that would benefit from parallelization. Quartz [1] is an older tool with similar goals; it computes normalized processing times on SMPs for functions using statistical sampling. TAU [29] is a flexible but complex parallel performance evaluation environment for multi-node HPC systems. Intel's Parallel Amplifier [12] and VTune Amplifier XE [13] allow software engineers to examine performance and scalability of programs and to visualize program hotspots and thread activity. McLaren's QProf [21] unites fine-grained timing measurements with estimates to provide detailed timings of multi-threaded program events. Tallent and Mellor-Crummey use sampling to identify program overhead and identify serialization in Cilk programs [33]. The Sun Studio performance tools identify lock contention, load imbalance, and memory contention in multi-threaded programs [15]. PGPROF from the Portland Group allows users to profile OpenMP and MPI programs and to analyze application scalability [32]. Additional OpenMP parallel performance tools include a runtime API for parallel profiling described by Hernandez et al. [10], and ompP [5]: a tool modeled off of mpiP [36] that identifies inefficient regions in OpenMP through source code instrumentation that counts OpenMP construct executions. The Pin binary instrumentation tool [26] can monitor a variety of performance metrics in parallel programs [3]. It is

best suited to applications where perturbation will not affect measurements, because it can cause significant timing overheads. Several parallelism analysis tools have been built on top of the Pin framework. PinPlay [24], for example, uses Pin to dynamically replay multi-threaded programs with the goal of fixing concurrency bugs. The CilkView Scalability Analyzer by He et al. [9] examines the dependencies in a program to estimate its parallelism using Pin to collect performance metrics serially. Finally, Moseley et al. [22] build a Pin tool that looks for loop behaviors that might indicate easy opportunities for parallelization.

LLVM Performance Tools. We build Harmony on top of the LLVM Compiler Framework. LLVM comes with several instrumentation features including block, edge and path profiling [19]. Like us, other teams have built custom instrumentation passes. For example, VMAD by Jimborean et al. [16] extends LLVM with a pass to support an instrumentation framework that can gather memory–access traces. Rane and Browne analyze memory traces via LLVM instrumentation [25], and Serebryany et al. use LLVM instrumentation for dynamic race detection [28].

Basic Block Profiling. Others have also profiled at the fine granularity of basic blocks. For example, Sherwood et al. [30] use Basic Block Vectors to identify similar intervals of execution in a program, and Smith's Pixie [31] tracer identifies basic block boundaries in MIPS code to count block executions and to monitor the number of branch instructions taken.

8 Conclusion

Like puzzles turned sideways, sometimes new perspectives can yield new insights. Unlike existing profiles which examine parallel programs from the perspective of a thread or process, parallel block vectors collect runtime statistics by basic block laterally by parallelism phase. Parallel block vectors show which parts of a program belong to the serial and parallel phases of execution and in what proportion. Collection of parallel block vectors is fast. This paper demonstrates Harmony, a compile-time instrumentation pass to collect runtime profiles with just 16-21% overhead. No manual code modification is required by the user, and profiles are architecture independent.

Fast collection coupled with detailed dynamic information about program behavior makes parallel block vectors broadly useful. This paper examined three ways parallel block vectors can shed light on hardware and micro-architectural design. The first identified basic blocks which do not fit the mold of Amdahl's pure parallelism and serialism and instead exhibit a mix of the two. The second demonstrated how parallel block vectors can uncover differences in program features at different degrees of parallelism. Finally parallel block vectors reveal that parallelism does not necessarily correlate with basic block execution frequencies.

9 Acknowledgements

This work has been made possible through the generous support of the National Science Foundation (CNS-1117135). We also thank Stephen Edwards and the anonymous reviewers for their comments on the manuscript.

References

[1] T. E. Anderson and E. D. Lazowska. Quartz: a tool for tuning parallel program performance. *SIGMETRICS Performance Evaluation Review*, 18:115–125, 1990.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 2010.

[3] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal. Analyzing parallel programs with Pin. *Computer*, 43(3):34 –41, 2010.

[4] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[5] K. Frlinger and M. Gerndt. ompP: A profiling tool for OpenMP. In *Proceedings of the International Workshop on OpenMP*, 2005.

[6] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 458–469, 2011.

[7] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Notices*, 17:120–126, 1982.

[8] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 205–216, 2010.

[9] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 145–156, 2010.

[10] O. Hernandez, R. C. Nanjegowda, B. Chapman, V. Bui, and R. Kufrin. Open source software support for the OpenMP runtime API for profiling. In *Proceedings of the International Conference on Parallel Processing Workshops*, ICPPW, pages 130–137, 2009.

[11] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41:33–38, 2008.

[12] Intel Corporation. Intel Parallel Amplifier 2011. `http://software.intel.com/en-us/articles/intel-parallel-amplifier/`.

[13] Intel Corporation. Intel VTune Amplifier XE. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[14] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, ISCA '07, pages 186–197, 2007.

[15] M. Itzkowitz and Y. Maruyama. HPC profiling with the SunStudio performance tools. In *Parallel Tools Workshop*, 2009.

[16] A. Jimborean, M. Herrmann, V. Loechner, and P. Clauss. VMAD: a virtual machine for advanced dynamic analysis of programs. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, 2011.

[17] D. Jones, Jr., S. Marlow, and S. Singh. Parallel performance tuning for haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, Haskell, pages 81–92, 2009.

[18] C. Kim, S. Sethumadhavan, D. Gulati, D. Burger, M. Govindan, N. Ranganathan, and S. Keckler. Composable lightweight processors. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, pages 381 –394, 2007.

[19] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 75–, 2004.

[20] A. D. Malony. Event-based performance perturbation: a case study. In *Proceedings of the ACM SIGNPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 201–212, 1991.

[21] G. McLaren. QProf: a scalable profiler for the Q back end. MIT PhD Thesis, 1995.

[22] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying potential parallelism via loop-centric profiling. In *Proceedings of the International Conference on Computing Frontiers*, CF, pages 143–152, 2007.

[23] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, pages 221–234, 2009.

[24] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 2–11, 2010.

[25] A. Rane and J. Browne. Performance optimization of data structures using memory access characterization. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 570 –574, 2011.

[26] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. PIN: A binary instrumentation tool for computer architecture research and education. In *Proceedings of the Workshop on Computer Architecture Education*, WCAE, 2004.

[27] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson. Programming model for a heterogeneous x86 platform. *SIGPLAN Notices*, 44:431–440, 2009.

[28] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov. Dynamic race detection with the LLVM compiler, 2011.

[29] S. S. Shende and A. D. Malony. The Tau parallel performance system. *International Journal of High Performance Computing Applications*, 20:287–311, 2006.

[30] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *SIGOPS Operating Systems Review*, 36:45–57, 2002.

[31] M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Department of Computer Science, Stanford University, 1991.

[32] STMicroelectronics, Inc. PGProf: parallel profiling for scientists and engineers, 2011. http://www.pgroup.com/products/pgprof.htm.

[33] N. R. Tallent and J. M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. *SIGPLAN Notices*, 44:229–240, 2009.

[34] Valgrind Developers. Cachegrind: a cache and branch-prediction profiler. http://valgrind.org/docs/manual/cg-manual.html.

[35] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. Taylor. Conservation cores: reducing the energy of mature computations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 205–218, 2010.

[36] J. Vetter and C. Chambreau. mpiP: Lightweight, Scalable MPI Profiling, 2011. http://mpip.sourceforge.net/.

[37] H. Zhong, S. Lieberman, and S. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proceedings of the Symposium on High Performance Computer Architecture (HPCA)*, pages 25 –36, 2007.