Embracing Heterogeneity with Dynamic Core Boosting

Hyoun Kyu Cho and Scott Mahlke
Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{netforce, mahlke}@umich.edu

ABSTRACT

Uniformly distributing parallel workloads amongst threads is an effective strategy for programmers to increase application performance. However, in any parallel segment, execution time is determined by the longest running thread. Even for embarrassingly parallel programs in the form of SPMD (single program multiple data), the threads are not perfectly balanced due to control flow divergence, non-deterministic memory latencies, and synchronization operations. Such an imbalance can be significantly exacerbated by performance asymmetry among cores, which is likely to exist in future generations of chip multiprocessors (CMPs) either for energy efficiency or due to process variation.

We propose Dynamic Core Boosting (DCB), a software-hardware cooperative system that mitigates the workload imbalance problem in performance asymmetric CMPs. Relying on dynamic voltage and frequency scaling to accelerate individual cores at a fine granularity, DCB attempts to balance the workloads by detecting and boosting critical threads. DCB coordinates its compiler and runtime to enable asymmetric CMPs to achieve near-optimal utilization of core boosting. The compiler instruments the program with instructions to give progress hints and the runtime monitors their execution, enabling DCB to intelligently accelerate selected threads within a total core boosting budget for better performance. On a simulated eight core system of varying frequency, our experiments using PARSEC benchmarks show that DCB improves the overall performance by an average of 33%, outperforming a reactive boosting scheme by an average of 10%.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Optimization*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Core Boosting, Critical Path Acceleration, Workload Balancing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the authors must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
CF'14, May 20 - 22 2014, Cagliari, Italy
Copyright is held by the owner/authors. Publication rights licensed to ACM.
ACM 978-1-4503-2870-8/14/05 $15.00.
http://dx.doi.org/10.1145/2597917.2597932

1. INTRODUCTION

Due to power dissipation limits and design complexity, the microprocessor industry has become less successful in improving the performance of monolithic processors, even with continued technology scaling. As a result, chip multiprocessors (CMPs) have grown into a standard for all ranges of computing from cellular phones to high-performance servers. Since CMPs require sufficient thread level parallelism (TLP) to benefit from the increased computing power, most performance-aware programmers face increasing pressure to parallelize their programs.

One lesson that programmers have learned from the long history of high performance computing is that increasing resource utilization results in better performance. As the multi-threaded programming model abstracts away the individual characteristics of each core, uniformly distributing workloads into threads has been considered an effective strategy to increase the utilization of CMPs.

Despite the best efforts of programmers to evenly divide workloads, it is very difficult, if not impossible, to perfectly balance workloads. Even for single program multiple data (SPMD) multithreaded workloads with embarrassing parallelism, there exists implicit software heterogeneity among threads due to control flow divergence, non-deterministic memory latencies, and synchronization operations. Such software heterogeneity sometimes inhibits the parallel programs from effectively utilizing a larger number of cores.

The performance asymmetry of cores can notably exacerbate workload imbalance, and it is highly probable that we will have asymmetry in the future generations of CMPs for several reasons. First, heterogeneous multicore systems have been introduced by many researchers for better performance [3, 18] or saving power [17]. Heterogeneous multicores are also an effective way to trade die area to higher energy efficiency [22], and some commercial products [13] have already started implementing such designs.

Increasing core-to-core process variation also creates performance asymmetry in CMPs [29]. Process variation is the phenomenon where the process parameters of transistors, such as effective gate length and threshold voltage, diverge from their nominal value affecting the maximum operable frequency. The amount of within-die process variation is growing, as integrated-circuit technology keeps scaling down the size of individual transistors. With the rapidly developing emphasis on power and energy efficiency, lower supply voltages are preferred by chip designers and this makes the variation problem worse. Future microprocessors are likely to be heterogeneous across the working frequency of individual cores, since making all cores run at the frequency of the slowest core loses too much performance in the presence of large process variation.

One possibility for dealing with performance asymmetry in CMPs is to place the burden of workload balancing on program-

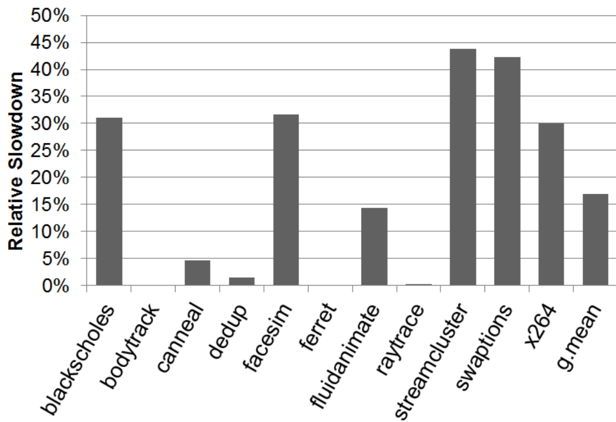

Figure 1: Slowdown caused by performance asymmetry.

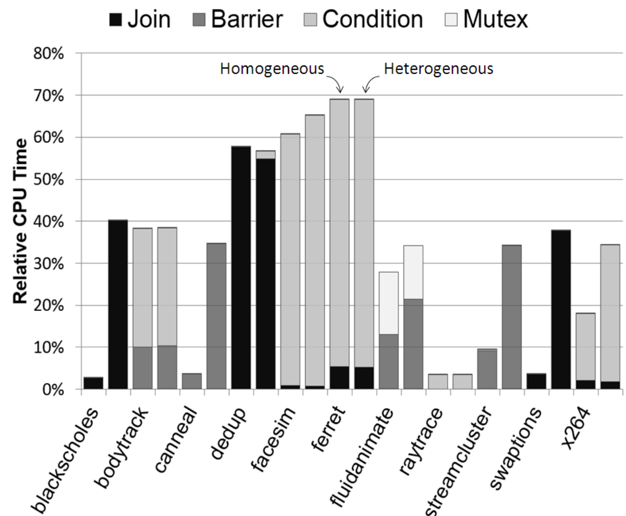

Figure 2: CPU time wasted for synchronization.

mers or compilers. However, parallel programming itself is already difficult enough for programmers. Even if we assume that it was possible for compilers to exploit the heterogeneity for workload balancing, the portability issue would prohibit them from generating the customized code for more than one specialized setting of heterogeneity. Furthermore, often the performance asymmetry caused by process variation cannot be determined at compile time because it may vary from one chip to another even for the same model processor.

In this paper, we propose Dynamic Core Boosting (DCB), a software-hardware cooperative system that mitigates the workload imbalance problem in performance asymmetric CMPs. DCB relies on the hardware capability of accelerating individual cores through dynamic voltage and frequency scaling (DVFS) at a fine granularity to balance the workload across the asymmetric cores by boosting critical threads. With the limited resource to boost a subset of cores, DCB orchestrates its compiler, runtime subsystem, and processor cores for near-optimal assignment of the boosting budget. First, a target program is analyzed and instrumented by the compiler to include the instructions that provide progress hints. At runtime, the execution of the program is monitored by the DCB runtime subsystem. Finally, DCB selectively boosts the critical threads by using the information gathered by the instrumented code and the DCB runtime subsystem.

This paper makes the following contributions:

- A theoretical background for the optimal assignment of core boosting.
- A cooperative system to balance workloads in asymmetric CMPs consisting of a compiler, runtime subsystem, and architecture.
- A novel mechanism to evaluate such systems with performance asymmetry and/or core boosting capability.

2. MOTIVATION AND BACKGROUND

While we can expect the performance asymmetry in CMPs to magnify the workload imbalance in multi-threaded programs, the exact effects on performance are not obvious. In this section, we present our motivation by showing the preliminary results on how much the asymmetry can affect the performance of multi-threaded benchmarks. Then, we provide the background of the hardware mechanism to accelerate individual cores.

2.1 Low Utilization of Asymmetric CMPs

We compare two simulated eight core systems to understand the performance impact of core asymmetry. The two systems work at the same average core frequency, but one has all eight cores operating at the same frequency and the other has varying frequencies. We assume a large variation in core frequencies ($\sigma/\mu = 30\%$, μ: mean, σ: standard deviation) as in Miller et al. [24], and the eight cores run at $(\mu-1.5\sigma)$, $(\mu-1.0\sigma)$, $(\mu-0.5\sigma)$, μ, μ, $(\mu+0.5\sigma)$, $(\mu+1.0\sigma)$, $(\mu+1.5\sigma)$, respectively. The details of evaluation methodology are explained in Section 5.

Figure 1 presents the slowdowns of the asymmetric system compared to the symmetric one for the PARSEC 2.1 benchmark suite [5]. Most of the benchmarks are configured to have the same number of worker threads as the number of cores, except for those with pipeline parallelism. *dedup* and *ferret* are set to have one thread per pipeline stage. *x264* spawns the number of worker threads equal to the number of frames and there is no trivial way to change it with the harness of the PARSEC benchmark suite.

Even though the two systems have the same average core frequency, we can see that many of the benchmarks experience significant slowdown. Several benchmarks such as *streamcluster* and *swaptions* suffer the slowdown close to the worst core frequency. Some others, i.e., *bodytrack*, *ferret*, and *raytrace*, show almost identical performance to the homogeneous system on the other hand. The geometric mean of the slowdown for all benchmarks is 17%.

In order to understand what causes more slowdowns for some benchmarks than the others, we measure how much portion of CPU time in parallel sections is wasted on each type of synchronization. Figure 2 presents the measured portions. For each benchmark, the left bar shows the CPU time spent running on the homogeneous cores and the right bar represents the time on the asymmetric CMP. As seen in the graph, the benchmarks use different types of synchronizations as their main mechanism to control parallel execution, and the impact of performance asymmetry varies depending on the dominant synchronization pattern.

The simplest method is to spawn threads to work independently and join them at the end. *blackscholes* and *swaptions* are in this category. Having similar structure, if the worker threads need to progress to the next stages together, they are synchronized with barriers. *canneal*, *fluidanimate*, and *streamcluster* use this type of synchronization patterns. For these two categories, the cores stay idle if their threads finish the tasks earlier than other cores, causing under-utilization of cores. Consequently, they are very likely to be affected by the asymmetry among cores.

Some benchmarks manage a pool of worker threads. When they

need to execute in parallel, the main thread distributes tasks to the threads in the pool. After they finish the tasks, they stay idle waiting for the next task. The worker threads are usually synchronized with condition variables. If the workload distribution is determined dynamically, e.g., *bodytrack* and *raytrace*, they are less susceptible to workload imbalance due to asymmetric cores. On the other hand, *facesim* is substantially affected by the asymmetry since the workload is equally divided once and assigned to the workers.

dedup and *ferret* adopt a pipeline parallel model. The worker threads run different stages of a pipeline and the data flows from one stage to another through a FIFO queue synchronized with condition variables. For this type of parallel program, the overall performance of the program is determined by the slowest stage. Accordingly, the performance is very sensitive to the stage-to-core scheduling for the asymmetric setting, but the average remains unchanged.

Finally, we see a great possibility of improving performance for asymmetric CMPs by balancing workloads. From the observations made above, many of the benchmarks are directly affected by the performance asymmetry. In addition, balancing the pipeline stages in the programs like *dedup* and *ferret* can yield performance benefits.

2.2 Core Boosting

Performance asymmetry among cores, combined with interthread dependencies formed by synchronization operations, causes a significant performance problem for multi-threaded programs as demonstrated above. We try to solve this problem by relying on the hardware capability of accelerating the subset of cores while staying in the power budget. Dynamic voltage and frequency scaling (DVFS) has been widely used for energy efficiency [1, 10]. Moreover, there have been several proposals that use dual power supplies for boosting individual cores [8, 24]. Dreslinski et al. [9] shows that very fast boosting transition ($<$ 10ns) can be achieved. Our system builds on such techniques for boosting cores at a fine granularity.

While the idea of adopting fast core boosting for mitigating performance bottlenecks or reducing performance heterogeneity is not new [8, 24], the main contribution of our work lies in how to assign core boosting for higher performance with the same power budget. We first provide the theoretical background for the optimal assignment of core boosting. In order to achieve a close to the optimum solution, we propose a system that coordinates the compiler, runtime, and processor cores.

One important point to notice is that our assignment techniques are not limited to the specific core boosting technology. Although we assume a dual V_{dd}-based core boosting to demonstrate the effectiveness of our techniques in this paper, our technique can be used in conjunction with any core acceleration mechanism with short enough transition time. Further differentiation from the previous proposals and more details of other feasible core boosting technologies are covered in Section 7.

3. CORE BOOSTING ASSIGNMENT

Given the core boosting capability and the limited boosting budget, how to assign the boosting budget is very important for overall performance. In this section, we show our core boosting assignment at an abstract level. At first, we describe the mathematical modeling of workload imbalance and core boosting. We then formulate core boosting assignment as an optimization problem and provide a theoretical solution. Finally, we explain our core boosting assignment algorithms for two commonly used parallelization practices: data parallel programs and pipeline parallel programs.

When programmers parallelize their compute intensive programs for better performance, they first have to decide how repeated computations can be divided into threads. If the computation is conducted on the multiple subsets of data and they can be potentially performed concurrently, data parallel structure is most commonly used. In this form of parallel programs, multiple worker threads are spawned to run same code on different, possibly overlapping, subsets of data. When some regions of code must be executed atomically, mutexes are used to guard the regions. In some cases, all worker threads should finish one phase of execution and be synchronized with each other before they start the next phase. Barrier waits are inserted between the phases for these cases.

For data parallel type of parallelism structure, software heterogeneity is implicit in the sense that worker threads run the same code. It does not always mean, however, that the amounts of computations are identical among the threads. Control flow divergence is the primary reason for such mismatch of computation. For example, if statements let different portions of code be executed depending on condition values. For some programs, even different number of loop iterations can be run depending on input data. Nondeterministic memory latencies are another important source of implicit software heterogeneity. Even though two threads are accessing the elements in the same array, one might hit and the other might miss in caches. Modern microprocessors usually have multiple levels of caches and accurately predicting the latency of each memory access is not possible. Lastly, synchronization operations also contribute to implicit software heterogeneity. For instance, when two threads are trying to acquire a mutex at the virtually same time, one might proceed immediately while the other waits until the mutex is released.

Another frequently used type of parallel structure is software pipelines. While the repeated computations can be executed concurrently in data parallel programs, some programs need to enforce orders among the computations performed on the different subsets of data. If different stages of computations can overlap preserving the orders, pipeline parallel structure is an option. For this type of parallel programs, multiple threads are spawned to execute the different stages of computations. Different stages are usually connected with FIFO queues and data elements flow from one stage to another through these queues. Condition variables are often used to synchronize the data flow.

Software heterogeneity is rather explicit in pipeline parallel programs, since different threads execute different codes. Since most modern microprocessors shows varying latencies depending on the types of instructions and the majority of them support out-of-order executions, statically balancing the execution time of different code is impossible even for homogeneous multicore processors. In addition, all sources of implicit software heterogeneity apply for pipeline parallel programs as well.

3.1 Modeling and Problem Formulation

Figure 3 depicts the modeling of workload imbalance and core boosting assignments with n cores. Without the loss of generality, this modeling assumes one workload for each core. If there are multiple threads running on a core, we can think of the total workloads of the threads as one workload. The assignment of core boosting can be changed after a certain predetermined amount of time, called a quantum. Note that this boosting quantum is much shorter than the traditional OS scheduling quantum. This is possible as core boosting take place with very short transition time as mentioned in the previous section. Then, $w_1, w_2, ..., w_n$ denote the number of quanta taken to run each workload without any boosting on Core_1, Core_2, ..., Core_n. Each core can be accelerated to a different extent for the boosted mode, and $b_1, b_2, ..., b_n$ are the amount

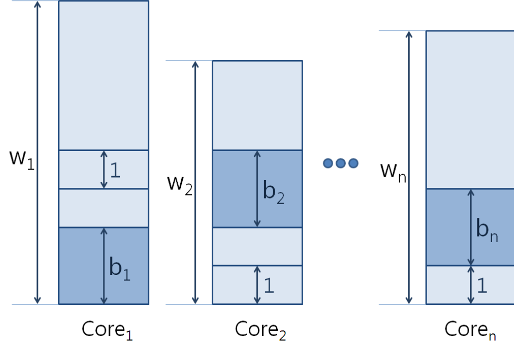

Figure 3: Modeling of workload imbalance and core boosting.

of acceleration. In addition, let $t_1, t_2, ..., t_n$ be the number of quanta where the boosting is assigned to each core.

Let us define the boosting budget, c, as the maximum number of cores that can be boosted at any quantum. For the best performance, c cores should be boosted every quantum, thus, it takes

$$T = \frac{1}{c} \times (t_1 + t_2 + ... + t_n) \tag{1}$$

boosting quanta to finish the execution. Moreover, $t_1, t_2, ..., t_n$ are bounded because a core can be boosted no more than once at any boosting quantum.

$$\forall 1 \le k \le n \ , \qquad 0 \le t_k \le T \tag{2}$$

The most important condition for this modeling to explain core boosting assignment is that every core must finish its workload within T quanta. For $\forall 1 \le k \le n$, $\mathrm{Core_k}$ runs t_k quanta boosted and $T - t_k$ quanta in normal mode, and it needs to finish its workload within T. Therefore, every t_k needs to satisfy the following inequality.

$$\forall 1 \le k \le n \ , \qquad (T - t_k) + b_k \times t_k \ge w_k \tag{3}$$

Since the number of boosted quanta for each core is an integer, core boosting assignment for the best performance is reduced to the integer linear programming [26] of minimizing T. Let us denote $P(w_1, w_2, ..., w_n)$ as the optimization problem of finding the minimal T and corresponding assignments $t_1, t_2, ...t_n$ when the workloads are $w_1, w_2, ..., w_3$.

3.2 Assignment for Data Parallel Programs

Although general integer linear programming is NP-hard, a solution can be quickly found with a greedy algorithm for our case. We will show that assigning the boosting budget to the cores with the largest remaining workload yields an optimal solution. We first prove the optimality of the greedy solution and then explain how we apply this to data parallel programs. For the simplicity of proof, c is assumed to be 1, but the same proof technique can be used for a larger boosting budget. The proof consists of two theorems.

THEOREM 1. *If w_p satisfies $max(w_1, w_2, ..., w_n) = w_p$, then there exists an optimal solution for $P(w_1, w_2, ..., w_n)$ where $t_p \ge 1$.*

PROOF. Suppose there exists an optimal solution, T' and $t'_1, t'_2, ..., t'_3$, where $t'_p = 0$. Since w_p is $max(w_1, w_2, ..., w_n)$ and $t'_p = 0$, the following can be derived from condition (3).

$$\forall 1 \le k \le n \ , \qquad T' \ge w_k \tag{4}$$

Then, let us find q such that $t'_q \ge 1$, and build another solution, T'' and $t''_1, t''_2, ..., t''_3$, by exchanging the values of t'_q and t'_p. Since we just exchanged two values, T'' remains the same as T'. From condition (4), this solution should also meets conditions (3). Therefore, T'' and $t''_1, t''_2, ..., t''_3$ is another optimal solution where $t'_p \ge 1$. □

THEOREM 2. *Let w_p satisfy $max(w_1, w_2, ..., w_n) = w_p$. If T' and $t'_1, t'_2, ..., t'_3$ with $t'_p \ge 1$ form an optimal solution for $P(w_1, w_2, ..., w_n)$, and T'' and $t''_1, t''_2, ..., t''_3$ form an optimal solution for $P(w_1 - 1, w_2 - 1, ..., w_{p-1} - 1, w_p - b_k, w_{p+1} - 1, ..., w_n - 1)$, then $T' = 1 + T''$.*

PROOF. Since T' and $t'_1, t'_2, ..., t'_3$ satisfy condition (3), we can show they also satisfy the following condition with a little manipulation.

$$\{(T' - 1) - t'_k\} + b_k \times t_k \ge (w_k - 1), \quad if \ \ k \ne p \tag{5}$$

$$\{(T' - 1) - (t'_k - 1)\} + b_k \times (t_k - 1) \ge (w_k - b_k), \quad if \ \ k = p$$

Thus, $(T' - 1)$ and $t'_1, ..., t'_{p-1}, (t'_p - 1), t'_{p+1}, ..., t'_n$ also form a solution for $P(w_1 - 1, w_2 - 1, ..., w_{p-1} - 1, w_p - b_k, w_{p+1} - 1, ..., w_n - 1)$. With the similar manipulation, we can show that $(T'' + 1)$ and $t''_1, ..., t''_{p-1}, (t''_p + 1), t''_{p+1}, ..., t''_n$ form a solution for $P(w_1, w_2, ..., w_n)$ as well. Now, if we assume $T' > 1 + T''$, it contradicts that T' is an optimal solution since $(1 + T'')$ is a solution. Likewise, assuming $T' < 1 + T''$ contradicts that T'' is optimal because $(T' - 1)$ is a solution. Therefore, $T' = 1 + T''$. □

The two proved theorems infer that boosting the core with the largest remaining workload at every quantum gives an optimal solution, hence the greedy algorithm will be optimal. Determining the remaining workload sizes at every quantum, however, is not possible in real systems. Consequently, we need a heuristic to decide which cores have the largest remaining workloads.

If we know the work progress ratio of each thread, we can approximately decide the thread with the least progress as the thread with the largest workload remaining. Although this heuristic is not always accurate, it works well when the threads are running similar amounts of workloads, which is usually the case for data parallel programs. As data parallel programs execute the same code for worker threads, we can instrument it to report work progress and assign a boosting budget to the cores with the least progress. The details of the program analysis and progress report instrumentation is explained in Section 4.3.

3.3 Assignment for Pipeline Parallel Programs

The heuristic used for data parallel programs does not work as well for pipeline parallel programs. It is primarily because pipeline parallel programs run different codes on different threads. It is difficult to measure progress consistently across threads running different codes. This makes it less likely that the thread with the least reported progress has the largest remaining work.

The synchronization pattern of pipeline parallel programs also makes it hard to apply the same technique. Multiple threads execute different stages of pipeline, and the data flows through the pipeline often using a FIFO queue. As it is difficult to perfectly balance workloads, some stages process data faster than the others. If one stage is significantly faster than its predecessor, the thread running the stage often waits on its input queue. Likewise, slow stages force their predecessors to wait. For this type of synchronization pattern, different stages make similar progress in terms of the number of data elements processed. Even though the same number of elements are remaining, however, faster stages have less workload than slow stages. This invalidates the greedy solution and requires us to use a different approach for pipeline parallel programs.

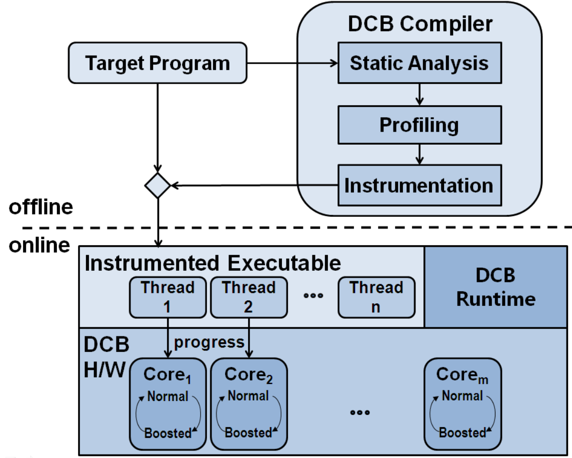

Figure 4: Dynamic Core Boosting system overview.

We adopt an epoch-based approach with the observation that the relative speeds among threads alter much more slowly than the boosting quanta. When we look at the ratio of time spent working and blocked at a coarser grain than a boosting quantum (100 - 1,000x), the ratio of each thread tends to stay constant for the longer period of time. Our approach exploits the trend by assuming that the workload size of the previous epoch closely represents the current epoch. The details of how the workload sizes are approximated at every epoch is described in Section 4.4

At the end of every epoch, the core boosting assignment is calculated for the next epoch. Since the assignment takes place at runtime and heavy computation can nullify the performance gain, we need a simple solution. Instead of solving the integer linear programming in Section 3.1, the integerality condition is ignored assuming epoch size is large enough so that linear programming relaxation yields a close approximation. A heuristic based on Simplex algorithm [25] is used to quickly find an approximation with a minimal amount of computation. Assuming the minimum value of T exists on one of the extreme points, condition (2) and (3) states

$$\forall 1 \leq k \leq n \ , \quad t_k = 0 \quad or \quad t_k = \frac{w_k - T}{b_k - 1} \tag{6}$$

As a heuristic, w_k is then compared to $max(\frac{w_1}{b_1}, \frac{w_2}{b_2}, ..., \frac{w_n}{b_n})$ and assigned to 0 if it is smaller. Finally, the rest of t_ks can be directly calculated according to equation (6).

4. DYNAMIC CORE BOOSTING

This section describes how our Dynamic Core Boosting system (DCB) coordinates the compiler, the runtime subsystem, and the underlying core boosting architecture to obtain improved performance by balancing workloads.

4.1 System Overview

Figure 4 represents the overview of DCB. The DCB compiler takes a target program as an input. It first analyzes the parallelism structure and the control flow of the program, and generates profiling code. The profiling code then runs with a training input and produces profile data. Additionally, the DCB compiler makes decisions based on the static analysis results and the profile data to instrument the program with progress monitoring code.

The generated executable runs on the DCB architecture along with the DCB runtime subsystem. In the DCB architecture, some cores can run in the boosted mode, which is faster than the normal mode. At every boosting quantum, the boosting manager in the DCB architecture decides which cores to run in the boosted mode while maintaining the boosting budget.

The instrumented code and the DCB runtime subsystem provide hints to the DCB architecture, with which the DCB architecture makes the boosting assignment decisions. For data parallel programs, the instrumented code reports the progress of each thread. At the end of every boosting quanta, the boosting manager chooses the threads with the smallest progress for boosting. DCB works differently for pipeline parallel programs. After every epoch, the DCB runtime subsystem calculates the desired boosting ratio among the threads to the DCB architecture, which stores the values for the next epoch. The boosting manager then probabilistically selects the cores to boost according to the boosting probability distribution.

4.2 DCB Architecture

While each core runs either in normal mode or boosted mode, it also takes hints and makes boosting assignments differently in two interface modes, namely progress mode and lottery mode, as briefly mentioned previously. The operating system takes this interface mode information with a flag for clone system calls when the threads are spawned. It stores the information and requests the DCB architecture to set the core in the proper mode every time a context switch occurs. In addition, the thread ID and the thread group ID are utilized by the DCB architecture when a thread is scheduled in.

The progress mode is mainly for data parallel programs. Each thread reports its progress to the DCB architecture. After every boosting quantum, the boosting manager chooses c threads with the least progress in the same thread group to be boosted, where c is the boosting budget assigned to the thread group. The DCB architecture provides two non-privileged instructions so that the instrumented code can report its progress without the intervention of the operating system. PROGRESS_STEP_FORWARD increases the progress counter of the core by one, and SET_PROGRESS_TO(value) sets the progress counter to value.

The lottery mode works in a slightly different way. Each thread does not directly interact with the DCB architecture. Instead, the DCB runtime subsystem sets the desired boosting ratio among threads after every epoch. The boosting manager probabilistically choose c cores based on the ratio distribution in a similar manner to how the Lottery Scheduler [30] allocates resources. Pipeline parallel programs use the lottery mode to implement the assignment algorithm explained in Section 3.3.

All per thread information needed for the boosting assignment is stored in thread boosting table, which is managed by the operating system in the same way as page tables. The operating system and the DCB architecture can both access and modify the values in the thread boosting table. Moreover, the DCB architecture includes a cache for the thread boosting table as TLB for the page tables.

4.3 DCB Compiler

The main goal of the DCB compiler is to instrument the target program with the progress reporting instructions so that the boosting assignment algorithm described in Section 3.2 yields near optimal performance. In order to do so, the DCB compiler works in three steps: static analysis, profiling, and instrumentation.

At first, the DCB compiler statically analyzes the parallelism structure and the control flow of the target program. For the parallelism structure it investigates the starting and ending points of parallel execution in the main thread and the highest level functions executed in parallel. For the majority of programs, they are thread spawning function calls, thread joining function calls, and functions passed over to the thread spawning function calls, respec-

```
01    : pthread_barrier_wait(barrier);
02(*): SET_PROGRESS_TO(0);
03(*): period = calc_period_L007(start,end);
04    : for( i = start ; i <= end ; ++i ) {
05    :     ...
06    :     compute1(...);
07    :     if(side_exit) {
08(*):         SET_PROGRESS_TO(MAX_L007);
09    :         break;
10    :     }
11(*):     if(((end - i) % period) == 0)
12(*):         PROGRESS_STEP_FORWARD;
13    : }
14    : compute2(...);
15(*): PROGRESS_STEP_FORWARD;
16(*): period = calc_period_L008(max);
17    : for( i = 0 ; i < max ; ++i ) {
18    :     compute3(...);
19(*):     if(((max - 1 - i)% period) == 0)
20(*):         PROGRESS_STEP_FORWARD;
21    : }
22    : pthread_barrier_wait(barrier);
```

Figure 5: Example of progress reporting instrumentation.

tively. For some programs the DCB compiler cannot accurately gather the information. For example, the DCB compiler might be unable to disambiguate the function pointers passed over to the thread spawning calls. Moreover, non-standardized task starting and ending functions are used when the program manages a thread pool and send tasks to the pool for parallel execution. In those cases, the DCB compiler relies on the programmers' annotation specifying the information.

Once the parallelism structure is determined, the DCB compiler analyzes the control flow of the code regions that can run in parallel. At the highest level, these sections are the functions passed over to the thread spawning calls and the region of the main threads between the starting and ending points of parallel execution. There could be function calls in these regions, and the DCB compiler follows the call graph to analyzes the callees in turn. It stops following the call graph if there is a call through an ambiguous function pointer or a cycle in the call graph. The barrier synchronization points are also included in the control flow information.

The DCB compiler generates the profiling code and runs it with a training input. It focuses on the loops in the parallel regions, using the control flow information gathered in the static analysis phase. The profiling code records the time spent in each loop and the iteration counts. Path profiling is also performed to discover the most frequent paths.

The last step exploits the profile data along with the static analysis results to instrument the code with the progress reporting instructions. In order to achieve the goal of the DCB compiler, all threads need to report progress at the points where they share the same progress ratio, regardless of what control path they take. One necessary condition is that all threads should go through the same number of progress reporting steps. It is straightforward for the counted loops with constant iterations. However, this is not always the case and other types of loops make this condition difficult to meet. In other words, naively incrementing a progress counter after every iteration does not work because the total iteration counts might vary across the threads even for the same loops depending on the input.

For the counted loops with input dependent iteration counts, the DCB compiler inserts the code to calculate the number of iterations needed to be executed for the next progress reporting right before entering the loop. This number is then used as a progress reporting period inside the loop. The DCB compiler also instruments loop side exits to set the progress counter to the final progress value of the loop. The DCB compiler does not instrument uncounted loops. If an uncounted loop in a parallel region takes too much time, it might hurt the workload balancing capability of DCB. However, it is a very rare case and the programmers can insert the progress reporting code by themselves or turn the loop into a counted loop. For instance, consider an uncounted loop traversing a linked list. It is very difficult for a compiler to decide the number of iterations before entering the loop. However, the programmer can possibly transform it to a counted loop by adding an element count variable in the list header.

Another requirement for the instrumented code is that the frequency of progress reporting should be adequate. If the reporting granularity is too coarse, the boosting manager cannot get enough information to decide the most lagging thread. It should not be too fine because the progress reporting instructions can incur excessive overheads for this case. The DCB compiler tries to insert progress reporting instructions so that the execution times between them are roughly constant. It estimates the execution time with the instruction counts for straight-lined code regions. In the case of loops, it uses the profile data to calculate the approximate execution time per iteration.

Figure 5 shows a simple example of how the instrumented code would look like in source level. The lines marked with an asterisk presents the code inserted by the DCB compiler. calc_period_L007() in line 3 and calc_period_L008() in line 16 are the inline functions generated by the DCB compiler. They calculate the number of loop iterations needed to be executed for the next progress reporting. Constant values cannot be used in the same place because of programs that have different number of iterations across the threads, since the total progress counts should be equal for all threads. The generated inline functions calculate the progress reporting period so that all threads go through the same number of progress reporting steps. Another point to notice is the line 8. For the threads that exits the loop before it finishes the total iterations, the DCB compiler sets the progress counter to the maximum progress of the loop.

4.4 DCB Runtime Subsystem

The most important role of the DCB runtime subsystem is to provide the desired boosting ratio to the DCB architecture when the threads are running in lottery mode. The DCB runtime subsystem is idle for the most of the time and wakes up after every epoch. It then reads the per thread values of the CPU cycles. The DCB architecture has the dedicated hardware counters for per core CPU cycles and the operating system manages the per thread values in the thread boosting table. The DCB runtime subsystem estimates the workload size of each thread by comparing the current per thread CPU cycles with the last value. Then it calculates the desired boosting ratio of the threads according to the assignment algorithm described in Section 3.3.

Although the DCB runtime subsystem can be implemented as a shared library, it is preferable for it to be part of the operating system because it needs fast accesses to the thread boosting table. Since the thread boosting table is protected from unprivileged accesses, the DCB runtime subsystem should go through the system call interface if it is implemented as a shared library. This can cause a performance problem if the epoch size is too small.

5. EVALUATION METHODOLOGY

As the system level interactions among threads are very important, the evaluation of DCB is different from the evaluation of other microarchitectural features. This difference makes the traditional

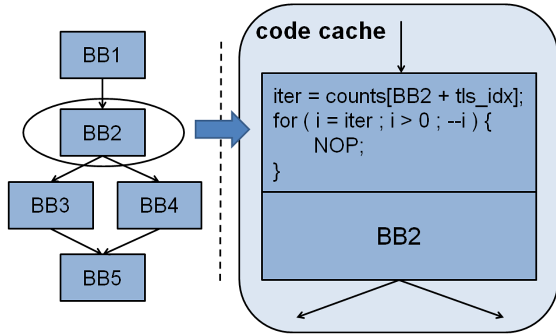

Figure 6: Core boosting emulation with dynamic binary translation.

evaluation approach of using cycle-accurate simulation an infeasible option for our purpose. DCB makes boosting assignment decisions based on the relative orders of thread progressions, and synchronization operations are critical to these orders. For instance, let us consider a situation where two threads are competing for a mutex. One of them is about to enter a long critical section and the critical section of the other is short. A slight difference of arrival time to the critical section can make a huge difference in the progress of the threads after they both exit from the critical sections. Moreover, even the execution path might change depending on the order of events [12]. Sampling [27] based simulation would not yield meaningful results as the interactions among threads are not considered. Trace-driven simulation that separates functional and timing simulation might not be accurate either.

Without sampling or trace-driven mechanisms, cycle-accurate simulators are too slow to evaluate the performance of DCB. The entire execution of the programs from the beginning to the very end must be measured since the interactions among threads are critical. This makes it very difficult, if not impossible, to test DCB on cycle-accurate simulators with realistic workloads. Therefore, we need a different approach.

In order to evaluate DCB in a reasonable amount of time while emphasizing on thread interactions, we use a dynamic binary translation (DBT) based emulation platform. For emulating diverse core speed for both performance asymmetry and core boosting, our platform slows down execution by adding extra instructions to each basic block. Figure 6 shows the conceptual diagram of this scheme. The iteration counts of the inserted *nop* loop decides how much the execution is slowed down. Since we need to vary the speed from thread to thread, Thread Local Storage (TLS) is used to store the index variable tls_idx. The transition between two different core speeds can be emulated by simply overwriting the value of this variable. The counts array is loaded to the memory before executing the program.

The key point for the accuracy of this evaluation scheme is that the amount of slowdown must be inversely proportional to the modeled core speed. We achieve this by judiciously deciding the iteration counts for every basic block and for every slowdown value. Our mechanism to decide the iteration counts is inspired by Eyerman et al. [11] which states that disruptive miss events such as cache misses and branch mispredictions result in characterizable performance behavior. The basic idea is that we can accurately dictate the iteration counts according to the required slowdown amount if we can measure the per basic block number of these disruptive events.

We choose the number of instructions, the last level cache misses, and the data TLB misses, since they showed the largest correlations with the CPU time of the programs in our measurement. Using hardware performance counters, we measure these values for various time periods during repeated execution of the benchmarks. We then model the relationship between the CPU time and those variables with linear regression based on the measurement.

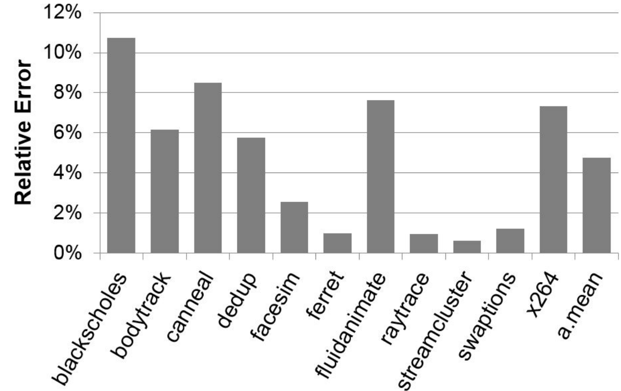

Figure 7: Errors in the simulated execution time of the performance asymmetry evaluation platform.

The hardware performance counters are also used for sampling the program counter values when the miss events occur. We collect the program counter samples to map the number of the miss events to each basic block. Assuming the sampling preserves the probabilistic distribution of the miss events, the numbers for the miss events per basic block can be calculated by projecting the sample distribution to the total number of miss events for the entire execution. Finally, the number of iterations per basic block and slowdown value are calculated according to the linear regression model along with the miss event numbers.

Except for the fact that each thread is slowed down, the execution on the evaluation platform is almost identical to running on native hardware. Since the threads actively interact with each other, the simulation errors caused by ignoring thread interactions can be minimized.

6. EXPERIMENTAL RESULTS

We first ascertain the validity of the evaluation platform by verifying the errors in the simulated execution time. Then, we use it to evaluate the performance improvement of DCB. We have built the evaluation platform on DynamoRIO [6], an open source dynamic binary translation system. The DCB compiler is implemented as an optimization pass for the LLVM compiler infrastructure [19], and we have implemented the DCB runtime subsystem as a shared library. All experiments are performed on a system with four 8-core Intel Xeon 2.26GHz processors with 24MB L3 cache, and the system has 32GB of main memory. We use the Pthreads implementation of PARSEC 2.1 benchmark suite [5], with *simlarge* workloads. *freqmine* is left out because it does not have a parallel version of Pthreads implementation. Although *vips* has Pthreads implementation, it is not used either since it works with GNOME Threads interface at the source code level. The current implementation of the DCB compiler needs source level interfacing with Pthreads for its static analysis. Each experiment represented is the average of the trials repeated at least ten times.

6.1 Accuracy of Evaluation Platform

We verify the accuracy of our evaluation platform by comparing the execution times with slowdown. Figure 7 shows the errors in the simulated execution time of the platform, dropping the sign for

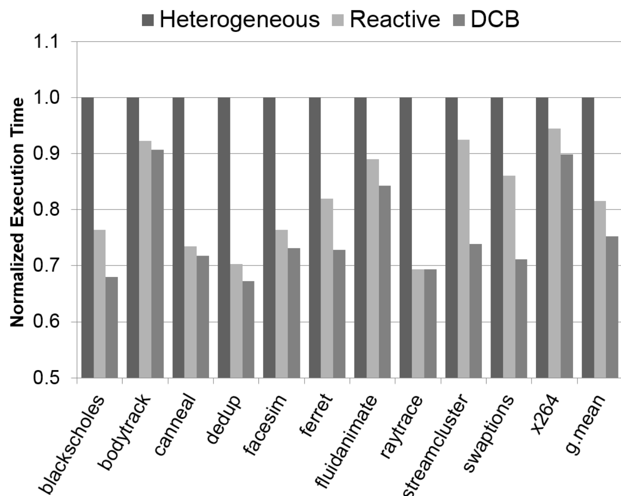

Figure 8: Normalized execution time of Heterogeneous, Reactive, and DCB.

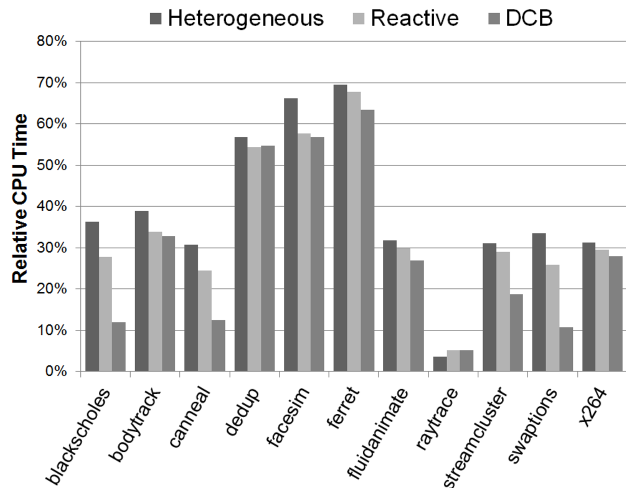

Figure 9: Synchronization overheads of Heterogeneous, Reactive and DCB.

negative values. For the experiments, we calculated the expected values from the simulated runs with 5x slowdown and compared them to the simulated runs with 10x slowdown. On average, our evaluation platform shows 4.8% of errors with the maximum of 10.8%. While our evaluation platform tries to closely match original execution using the inferred linear regression model and the per basic block hardware counter statistics, the main source of error is the difference between the original instructions and the extra instructions instrumented. Despite the fact that it does not perform the detailed microarchitectural simulation, however, it is quite accurate. More importantly, it enables us to run the programs on realistic inputs without sampling while correctly maintaining interdependencies arising due to synchronizations.

6.2 DCB Performance Improvement

Using the DBT-based performance asymmetry evaluation platform, we evaluate the performance improvement of the DCB system. The underlying asymmetric CMP is assumed to be identical to the one used in Section 2.1. The standard deviation (σ) of the core frequencies is 30% of the average (μ), and the eight cores run at the frequencies of ($\mu - 1.5\sigma$), ($\mu - 1.0\sigma$), ($\mu - 0.5\sigma$), μ, μ, ($\mu + 0.5\sigma$), ($\mu + 1.0\sigma$), ($\mu + 1.5\sigma$), respectively. As the current generation of AMD processors [1] already have per-core DVFS capable of operating at 20 - 30% higher frequencies than the nominal frequencies, we use the acceleration value of 1.5x assuming fast switching ($<$ 10ns) with dual supply voltage rails. We use $c = 1$ for the boosting budget, which means one core can be boosted at any moment. We use the asymmetric CMP with no boosting, *Heterogeneous*, as a baseline. For the fairness of comparison, the frequencies of *Heterogeneous* is set to be higher than the underlying cores for the boosting schemes so that its average core frequency is equal to the boosting schemes. Although we cannot directly measure power consumption due to the limitation of our evaluation platform, we keep the power budgets of boosting schemes as close to the baseline as possible in this way.

We also compare DCB to a reactive boosting scheme, *Reactive*, where the priority of the threads is managed in the same way as a state-of-the-art reactive core acceleration scheme, *Booster SYNC* [24]. In *Reactive*, a thread can be in one of the three priorities: *blocked*, *normal*, and *critical*. The default priority is *normal* and this changes to *blocked* when the thread is waiting for either a mutex, a condition variable, or barrier. The priority is promoted to *critical* if the thread acquires a mutex. *Reactive* always prefers the thread with higher priority. When there are multiple threads with the same highest priority, *Reactive* assigns boosting in a round robin manner.

Figure 8 shows the normalized execution time of *Heterogeneous*, *Reactive*, and *DCB*. *DCB* achieves performance improvement over both *Heterogeneous* and *Reactive* across all of the benchmarks. On average, the performance gain of *DCB* over *Heterogeneous* is 32.9%, outperforming *Reactive* by 10.3%. As expected from the preliminary analysis in Section 2.1, *DCB* is most effective for the benchmarks having thread join or barriers as the primary synchronization method, as in *blackscholes* and *streamcluster*. Interestingly, both *Reactive* and *DCB* present substantial performance improvement even for the benchmarks with dynamic workload distribution, such as *bodytrack* and *raytrace*, mainly due to the sequential regions. For the sequential portions of executions, both *Reactive* and *DCB* can concentrate the boosting budget to the only working thread yielding better performance than *Heterogeneous*.

In order to better understand the workload balancing capability of *DCB* without the effect of accelerating sequential region, we also measure the CPU time wasted for synchronization operations in the same way as in Figure 2. Figure 9 presents the CPU time portion for synchronizations. From this graph, we can confirm that *DCB* is very effective in balancing workloads and reducing the synchronization overheads, for data parallel programs such as *blackscholes* and *streamcluster*. We can also see that *DCB* can reduce the synchronization overhead of pipeline parallel programs like *ferret*. Note that this graph shows the ratio of synchronization overheads to the total CPU time of parallel execution. Since the execution time is significantly reduced for benchmarks like *dedup* and *ferret*, the workload balancing effect is actually greater than it looks in the graph.

Figure 10 illustrates how *DCB* outperforms the other schemes. In this figure, X-axis presents the time scale normalized against the finishing time of the last threads of *Heterogeneous*, and Y-axis is for the number of threads that have finished their tasks. Therefore, if the line hits the ceiling earlier, better performance was achieved. As expected, *DCB* shows the best performance among all the schemes. An interesting point to note is that *DCB* loses to the other schemes until the sixth thread finishes its task. This shows that *DCB* assigns the boosting budget in a way closer to the optimum. In other words, *DCB* saves the boosting budget from already fast threads and assign them to the lagging threads, reducing the workload imbalance. For

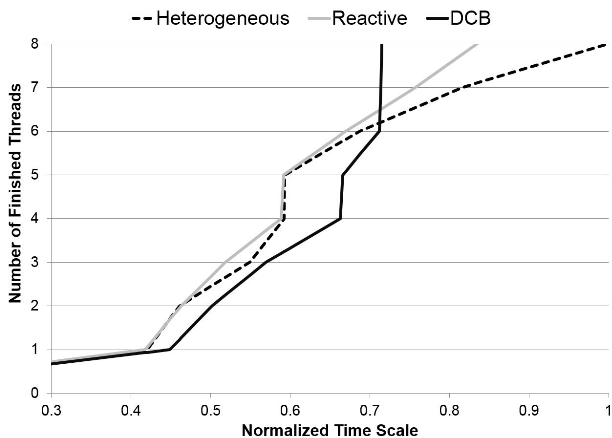

Figure 10: Arrival time of each thread for blackscholes.

this reason, the slope of the *DCB* line becomes steeper as it gets to the end. For DCB, only the last three threads are finishing their tasks approximately at the same time, and this is because the boosting budget is not enough to balance all of the threads. If *DCB* had more boosting budget, it would have the almost vertical fraction of the line from an earlier point.

Another point to notice in Figure 10 is that *Reactive* starts almost identically with *Heterogeneous* and gains a slightly steeper slope than *Heterogeneous*. The reason is that *Reactive* is indeed reactive. *Reactive* does not discriminate threads before some of them reach synchronization operations. So, it starts identical with *Heterogeneous* fairly distributing the boosting budget to all cores. This is also why *Reactive* beats *DCB* in the beginning. Moreover, the *Reactive* line is slightly steeper than *Heterogeneous* because it starts concentrating the boosting budget by not assigning it to the idle cores.

7. RELATED WORK

In this section, we first survey previous work that suggests performance asymmetry in CMPs. Since DCB is not limited to one type of core boosting mechanism as mentioned before, we then review per-core performance adaptation technologies that could possibly be used for core boosting. Finally, we study the previous proposals for assessing thread criticality and differentiate DCB from them.

7.1 Performance Asymmetry in CMPs

There have been numerous prior works that motivate inherent performance asymmetry in CMP designs. Several of them [3, 18] are proposed for better performance, and some others [17] show asymmetry is beneficial to reduce power consumption. Asymmetric CMPs have been demonstrated to be effective for alleviating serial bottlenecks [14, 28, 16]. In consequence, some commercial products [13] have started adopting the trend.

Increasing within-die process variation in near-future technologies also demands performance asymmetry even in homogeneous CMP designs. Because of process variation, Teodorescu et al. [29] claims that it is no longer accurate to think of large CMPs as homogeneous systems. Furthermore, low voltage chips aggravate the impact of process variation, and maintaining homogeneity by operating at the frequency of the slowest core severely lowers performance [23].

7.2 Dynamic Adaptation of Core Performance

Dynamic voltage and frequency scaling (DVFS) is a widely used technique for dynamic per-core performance adaptation [15, 10] and some AMD commercial processors support per-core DVFS [1]. However, off-chip regulator-based DVFS incurs intolerable scaling overheads (tens of microseconds) for our purpose. On the other hand, DVFS using on-chip regulators has much shorter transition time but suffers from low efficiency of the regulators.

Miller et al. [24] and Dreslinski [8] recently proposed the use of dual-voltage rails for fast adaptation of per-core performance. In addition, Dreslinski et al. [9] confirmed that very short transition time ($<$ 10ns) is achievable with a new circuit technique. We assume this technique to demonstrate the effectiveness of the DCB system.

Another feasible option for the underlying mechanism of core boosting is adapting hardware resources of cores. Composite Cores [22] integrates two different types of computing engines and achieves high performance and energy efficiency. It also shows that fine-grained (quantum length of 1000 instructions) dynamic per-core performance adaptation is possible.

While Composite Cores adapts in-core hardware resources, Illusionist [2] uses another core to boost cores. Illusionist consists of many lightweight cores and a small number of aggressive cores, and aggressive cores are used to accelerate the execution of the lightweight cores by providing execution hints, running ahead of them.

7.3 Thread Criticality Assessment

Thread Criticality Predictor (TCP) [4] identifies thread criticality based on memory hierarchy statistics using hardware counters. It increases energy efficiency by scaling down the frequency of non-critical threads or improve performance by task stealing from critical threads. Although TCP shows high accuracy (average of 93%), it is not suitable for our purpose of balancing workloads for asymmetric CMPs. For example, consider two perfectly identical (including cache misses) threads running on two cores with different frequencies. In the middle of the workloads, TCP would assess the criticality of faster thread higher than the slower thread because the faster thread would have more misses to the point.

Prior work has suggested using barrier synchronizations for thread criticality prediction for saving energy either by transitioning into low power modes after reaching a barrier or by scaling down the voltage and frequency of non-critical threads. Liu et al. [21] and Thrifty Barrier [20] differ from our work as they try to predict the arrival time to the next barrier based on history while DCB only needs to decide lagging threads for data parallel programs. Meeting Points [7] is similar to our work considering that it employs instrumenting programs with special instructions for monitoring progress. However, it only works for regular parallel loops with identical iteration counts across all threads, as opposed to DCB which can handle not only the loops with varying iteration counts but also the threads with different code.

Accelerating Critical Sections (ACS) [28] and Bottleneck Identification and Scheduling (BIS) [16] also use special instructions for detecting bottlenecks. Especially, BIS measures the number of cycles spent by threads waiting for each bottleneck and accelerates the bottlenecks responsible for the highest thread waiting cycles. The primary difference of ACS and BIS from our work is that they work in coarser granularity since they rely on thread migration to accelerate bottlenecks.

The most closely related work to DCB is Booster [24], where it also tries to balance multi-threaded workloads using core boosting. They propose two boosting algorithms: Booster VAR and Booster

SYNC. Booster VAR balances the CPU cycles spent by each thread and Booster SYNC improves it by taking priority hints from synchronizations. The most important difference between Booster and DCB is that Booster is reactive. Even Booster SYNC does not discriminate threads until they reach synchronization operations. Therefore, it cannot address implicit software heterogeneity caused by control flow divergence and non-deterministic memory latencies. Similarly, it is not well-suited for pipeline parallel programs. Even though different stages are heavily biased, Booster gives up the chance of balancing them until some of them get blocked for synchronizations. Conversely, DCB is proactive handling software heterogeneity very well. Finally, it is not trivial to extend Booster for other types of asymmetric CMPs or core boosting mechanisms, since it uses the core frequency values for balancing cycles. Meanwhile, DCB is applicable to them without any modification for data parallel programs and it only needs relative acceleration ratio for pipeline parallel programs.

8. CONCLUSION

This paper investigated the elimination of workload imbalances in performance asymmetric CMPs by relying on the hardware capability to accelerate individual cores at a fine granularity. We proposed Dynamic Core Boosting (DCB), a software-hardware cooperative system that balances the workloads by boosting critical threads. DCB coordinates its compiler, runtime, and processor cores, for near-optimal assignment of core boosting. The DCB compiler instruments target programs with instructions to give progress hints. The DCB runtime subsystem monitors their execution, enabling intelligent assignment of the boosting budget for better performance. On a simulated eight core system of varying frequency, our experiments using PARSEC benchmarks showed that DCB improves the overall performance by an average of 33%, outperforming a reactive boosting scheme by an average of 10%.

9. ACKNOWLEDGMENTS

Much gratitude goes to the anonymous reviewers for their excellent feedback on this work. This research is supported in part by the National Science Foundation under grant CNS-0964478 and by the Defense Advanced Research Projects Agency under the Power Efficiency Revolution for Embedded Computing Technologies (PERFECT) program. We would also like to thank the CCCP group members for their valuable comments.

10. REFERENCES

[1] AMD. *AMD Family 10h Server and Workstation Processor Power and Thermal Data Sheet*, June 2010. http://support.amd.com/us/Processor_TechDocs/43374.pdf.

[2] A. Ansari, S. Feng, S. Gupta, J. Torrellas, and S. Mahlke. Illusionist: Transforming lightweight cores into aggressive cores on demand. In *Proc. of the 19th International Symposium on High-Performance Computer Architecture*, pages 436–447, 2013.

[3] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 506–517, 2005.

[4] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 290–301, 2009.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[6] D. L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institude of Technology, Sept. 2004.

[7] Q. Cai, J. Gonzalez, R. Rakvic, G. Magklis, P. Chaparro, and A. Gonzalez. Meeting points: Using thread criticality to adapt multicore hardware to parallel regions. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 240–249, 2008.

[8] R. G. Dreslinski. *Near Threshold Computing: From Single Core to Many-Core Energy Efficient Architectures*. PhD thesis, University of Michigan, 2011.

[9] R. G. Dreslinski, B. Giridhar, N. Pinckney, D. Blaauw, D. Sylvester, and T. Mudge. Reevaluating fast dual-voltage power rail switching circuitry, June 2012. In 10th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD) in conjunction with ISCA 39.

[10] S. Eyerman and L. Eeckhout. Fine-grained dvfs using on-chip regulators. *ACM Transactions on Architecture and Code Optimization*, 8(1), 2011.

[11] S. Eyerman, L. Eeckout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems*, 27(2), 2009.

[12] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *Proc. of the '93 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 146–157, 1993.

[13] P. Greenhalgh. Big.little processing with arm cortex-a15 & cortex-a7: Improving energy efficiency in high-performance mobile platforms, Sept. 2011. http://www.arm.com/files/downloads/big.LITTLE_Final.pdf.

[14] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(1):33–38, 2008.

[15] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, pages 347–358, Dec. 2006.

[16] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–234, 2012.

[17] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 81–92, Dec. 2003.

[18] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, page 64, 2004.

[19] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[20] J. Li, J. F. Martinez, and M. C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *Proc. of the 10th International Symposium on High-Performance Computer Architecture*, page 14, 2004.

[21] C. Liu, A. Sivasubramaniam, M. Kademir, and M. J. Irwin. Exploiting barriers to optimize power consumption of cmps. In *Proc. of the 19th Int'l Parallel and Distributed Processing Symposium*, page 5a, 2005.

[22] A. Lukefahr, S. Padmanabha, R. Das, F. Sleiman, R. Dreslinski, T. Wenisch, and S. Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proc. of the 45th Annual International Symposium on Microarchitecture*, 2012.

[23] T. Miller, R. Thomas, and R. Teodorescu. Mitigating the effects of process variation in ultra-low voltage chip multiprocessors using dual supply voltages and half-speed units. *Computer Architecture Letters*, 11(2):45–48, 2012.

[24] T. N. Miller, X. Pan, R. Thomas, N. Sedaghti, and R. Teodorescu. Booster: Reactive core acceleration for mitigating the effects of process variation and application imbalance in low-voltage chips. In *Proc. of the 18th International Symposium on High-Performance Computer Architecture*, pages 1–12, 2012.

[25] K. G. Murty. *Linear Programming*. Wiley, 1983.

[26] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 1998.

[27] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 244, 2003.

[28] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 253–264, 2009.

[29] R. Teodorescu and J. Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 363–374, June 2008.

[30] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, 1994.