A Case for a More Effective, Power-Efficient Turbo Boosting

SUSHANT KONDGULI and MICHAEL HUANG, University of Rochester

Single-thread performance and throughput often pose different design constraints and require compromises. Mainstream CPUs today incorporate a non-trivial number of cores, even for mobile devices. For power and thermal considerations, by default, a single core does not operate at the maximum performance level. When operating conditions allow, however, commercial products often rely on turbo boosting, which temporarily increases the clock frequency to increase single-thread performance. However, increasing clock speed may result in a poor performance return for invested energy. In this article, we make a case for a more effective boosting strategy, which invests energy in activities with the best estimated return. In addition to running faster clocks, we can also use a look-ahead thread to overlap the penalties of cache misses and branch mis-predicts. Overall, for similar power consumptions, the proposed adaptive turbo boosting strategy can achieve about twice the performance benefits while halving the energy overhead.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; Heterogeneous (hybrid) systems; • **Hardware** → Chip-level power issues;

Additional Key Words and Phrases: Turbo boosting, implicit parallelism, helper threads

ACM Reference format:

Sushant Kondguli and Michael Huang. 2018. A Case for a More Effective, Power-Efficient Turbo Boosting. *ACM Trans. Archit. Code Optim.* 15, 1, Article 5 (March 2018), 22 pages.

<https://doi.org/10.1145/3170433>

1 INTRODUCTION

The central goal of processor design has always been to best utilize available resources to achieve maximum performance. One challenge is that the operating situation is unpredictable: Sometimes there are multiple ready software threads, and then improving system performance may be done by executing more of them in parallel; other times, improving single-thread performance is the only way to improve that of the whole system.

In general, single-thread performance and throughput pull the resource allocation decision in different directions and require compromises. For instance, mainstream CPUs today incorporate a non-trivial number of cores, even for mobile devices, and for power and thermal considerations, by default, a single core does not operate at the maximum performance level. One general way of addressing the difference in resource demand is to allow dynamic reallocation. Simultaneous multithreading and turbo boosting are two examples widely adopted in high-end commercial products [40, 66]. In turbo boosting, the system reallocates unused thermal and power headroom from idle cores to boost the frequency of a busy core to improve its single-thread performance.

Authors' addresses: S. Kondguli and M. Huang, University of Rochester, CSB 406, 160 Trustee Road, Rochester, NY 14627; emails: {sushant.kondguli, michael.huang}@rochester.edu.

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 ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 ACM 1544-3566/2018/03-ART5 \$15.00

<https://doi.org/10.1145/3170433>

While the general idea of dynamic resource allocation is sound, experience has shown that increasing clock speed has not been a very effective approach to boosting performance in general—this is partly why chip operating frequency has been relatively static for more than a decade now. In this article, we make a case for a more effective boosting strategy, which is cognizant of the behavior of the code and invests energy in activities with the best estimated performance benefits. In some cases increasing the core frequency is the right action. Other times, especially in the presence of significant cache misses and/or branch mispredictions, we can dynamically increase the strength of branch prediction and prefetching by executing a look-ahead thread on the idle core. We will show that both the architectural and software support for such a decoupled look-ahead (DLA) system are practical. We also find that simple controller logic can do a reasonably good job at managing such an adaptive turbo boosting system. Overall, the proposed design is significantly more effective in improving performance and is much more energy efficient: Compared to frequency-only boosting, the adaptive design achieves about twice the performance benefits at half the energy overhead. To summarize, we make the following contributions in this article:

- (1) Even though turbo boosting has been studied extensively, we proposed a relatively simple addition that makes it *significantly* more efficient than the state of the art.
- (2) We present an architectural support for DLA, purposefully kept at a minimum complexity, making it a compelling feature to add to modern architectures.
- (3) We performed detailed, multifaceted analyses to permit a reader to understand clearly where the benefits come from.

2 BACKGROUND AND RELATED WORKS

2.1 Turbo Boosting Technology

Commercially available conventional turbo boosting, via frequency and voltage scaling, achieves moderate performance gain at the cost of disproportionate power and energy consumption [17, 28]. Turbo boosting is often more helpful for CPU bound applications and naive use can significantly increase power consumption without comparable performance gains. A recent study found that turbo mode is also inefficient for workloads that exhibit interference for shared resources [46]. Recently proposed *AutoTurbo* and *Cooperative Boosting* techniques try to improve the power and energy efficiency of baseline Turbo Boosting [46, 56].

Another form of turbo boosting is *computational sprinting* in which a chip temporarily exceeds its sustainable thermal power budget to provide instantaneous throughput, after which the chip must return to nominal operation to cool down [58, 59]. Computational sprinting is useful for the class of applications that benefit from instantaneous burst and often becomes challenging when there are multiple agents to sprint in a CMP like setup. Fan et al. treat the sprinting management problem as a repeated game that produces an equilibrium to improve the overall throughput [29].

There has been plethora of work on dynamic voltage frequency scaling (DVFS) to improve power, performance, energy, or any combination of them. Efficient management of DVFS knobs across multiple voltage and frequency domains becomes challenging, therefore, recent DVFS proposals focus primarily on managing multiple V/F domains to improve the overall efficiency [20, 68, 70, 71]. Program behavior-specific techniques also have been proposed to improve the baseline DVFS [37]. Presence of a realistic memory system presents additional challenges for DVFS management; *CoScale* and *CRIT+BW* proposals try to improve the performance and overall efficiency for such systems [23, 51].

Compared to aforementioned DVFS management techniques, our technique is not about better ways of controlling frequency and voltage knobs. Instead, the key focus is *whether* to apply it or other forms of performance boosting.

2.2 Helper Threading

Various flavors of helper threading have been proposed in the past. Micro helper threading [4, 16, 21, 30, 43, 47, 48, 52, 53, 60, 73, 74] launches short threads ahead of potentially stalling instructions to avoid performance bottlenecks. These threads include specific instructions (often identified via lengthy manual tuning) required to precompute address or branch direction in a timely manner. Also, these threads are more suited to thread contexts that are designed specifically for them [16].

Another body of work attempts to provide a hardware solution to helper threading [14, 15, 18, 27, 36, 41, 42, 44, 54, 65, 67]. These proposals either start from an out-of-order core design and add hardware components to alleviate performance in case of stalls or use a modified in-order core with additional hardware components to optimally extract ILP. Most of these techniques generate a reduced copy of the original program in runtime. Although, the amount of information available is limited at runtime and this limits their performance.

Idle core in a multicore system can also be used to execute a reduced copy of the original program on a separate thread context [5, 11, 31, 33, 34, 50, 69, 72]. Many of these proposals target the same performance hurdles but differ in the way they accelerate the look ahead core. We call this flavor of helper threads DLA. Unlike micro helper threads, a decoupled look-ahead thread is a continuous process and requires less micromanagement from the main thread and therefore presents a powerful option for turbo boosting without slowing down the main thread.

3 OPPORTUNITY ANALYSIS

When operating conditions allow, a number of things can be done to improve the execution speed of the processor, albeit at the expense of extra energy expenditure. Increasing clock frequency is straightforward in that commercial processors already come with the capability to dynamically adjust clock frequency and voltage supply. However, the need to also increase the supply voltage can result in disproportional energy overhead for the performance gain.

Another way to improve performance is to reduce the number of stall cycles. Conceptually, this can be done by activating more powerful branch predictors or prefetchers or other means of increasing the effort to expose and exploit inherent parallelism. Depending on the execution behavior, targeting these stall cycles may offer a better opportunity to improve performance than boosting the clock frequency.

To have a more concrete understanding of the potential benefits, we take SPEC CPU benchmarks and compare a few configurations. Specifically, we measure the speedup obtained through an increase of clock frequency (by 33%) and elimination of branch mispredicts, L2 cache misses, or L1 cache misses. Since some results are from idealized configurations, they are only meant as rough characterizations of the potentials.

In Figure 1, the y-coordinate of each application shows the speedup achieved by a particular boosting configuration. To show the correlation between boosting effect and baseline performance, the x-coordinate shows the average IPC of that application (over the entire observation window in the baseline configuration). Note that showing such correlation does not imply one metric is a function of the other. Linear regression results are shown as solid lines to indicate some first-order trend.

We can observe several things from the experiments. Not surprisingly, different boosting mechanisms do not have the same effect on all applications. Also, boosting clock frequency is complementary to other type of boosting (suggested by the idealizations). Indeed, the effect of increasing clock frequency has a positive correlation with baseline performance while the other boostings have a negative correlation. In other words, increasing the clock frequency tends to be less effective for slower applications. Intuitively, those applications are slow often due to poor cache

Fig. 1. Performance improvements under different boosting scenarios: (a) boosting clock frequency by 33%; (b) eliminating all branch mispredictions; (c) eliminating L2 cache misses; and (d) eliminating all L1 cache misses.

Fig. 2. Varying behavior within a single application (lbn). Various statistics (L1 and L2 cache misses, and branch mispredicts measured in misses/mispredicts per kilo instruction, MPKI) for each dynamic instance of different loops. Dots of the same color represent instances of the same loop. While different instances of the same loop cluster together, demonstrating similar characteristics, different loops are quite distinct.

behavior and hence elevated memory accesses. Increasing core clock frequency has limited benefit when memory access becomes the limiting factor. Instead, these applications tend to enjoy significant speedups from better caching behavior.

Clearly, when deciding on the proper form of boosting, we need to take into account the program behavior. Even within one program, the behavior also varies, demonstrating what is often called *phase behavior* [2, 10, 12, 24, 26, 49, 61, 63, 64]. In Figure 2, we show a few common statistics about program behavior measured over different loops of a random example application. The result illustrates the behavior difference even within the same program.

To sum up, there is significant variation of program behavior both within and across different applications. As a result of this variation, the best mechanism to boost system performance (when allowed additional energy budget) may be very different. A fixed policy of always investing that energy budget in clock frequency boosting is likely to be a suboptimal strategy. A sensible turbo boosting strategy should also include mechanisms that targets stall cycle reduction and a controller that can make judicious selections. We discuss our proposed designs next.

4 ARCHITECTURE DESIGN

The concept of a dual-threaded execution mode that we call decoupled look-ahead is not new. We keep this concept (which has many different implementation proposals) but provide a simple implementation. We first describe the underlying components in Section 4.1.

Additionally, we propose a control logic to enable what we call Adaptive Turbo Boosting (ATB) to achieve the goal of maximizing performance gain for the invested energy. As it turns out,

energy expenditure is strongly correlated with performance benefit: The best performing boosting configuration also costs the least energy in a vast majority of the phases.¹ Our design, therefore, simply picks the best performing configuration for each phase of execution based on their code structure. We discuss the controller in Section 4.2.

4.1 Dynamic Boosting Mechanisms

Increasing clock speed is a readily available mechanism in commercial processors. Dynamically boosting the performance of cache and branch predictor is less straightforward. One can imagine reconfigurable branch predictors [19, 38] and prefetchers [75, 76, 77]. But in this article, we focus on using the available idle core, which is often (though not always) the source of available power and thermal headroom to apply boosting.

Prior Art: Indeed, a number of designs have been proposed to use an otherwise idle core to perform some kind of look-ahead action to assist the main core. The key challenge is to make the look-ahead action fast enough that there will be sufficient assistance rendered to the actual execution thread. A number of techniques are proposed to achieve that by reducing logic complexity of the core design [8, 9, 50], skipping predicted dead instructions [57, 69], skipping long-latency accesses [11, 72], optimistic over-clocking [33], and using a special-purpose look-ahead thread (with special various speculative optimizations) [5, 7, 31, 32, 55].

Note that another large body of work uses more specialized micro threads to perform more targeted look-ahead (e.g., References [4, 16, 21, 25, 30, 47, 52, 60, 73]). The support of the execution of these special micro threads tends to be an integrated part of the core, and thus does not lend directly to the situation of leveraging an idle core in a multicore architecture. Nevertheless, the overall take-away point is that there are numerous ideas about software-directed look-ahead to reduce stalls. It is not difficult to imagine a future processor to support some combination of the proposed features to allow dynamic activation of look-ahead thread as a performance boosting mechanism.

Overview of Support: In this article, we use a similar setup along the lines of a number of prior proposals [7, 11, 31, 57, 69, 72]. On top of a generic multi-core architecture, we require the following architectural support, ordered from least to most special-purpose:

- (1) Containment of speculation: The look-ahead thread usually involves speculative optimizations and thus can not be allowed to update the architectural state. The support is simple as most of the state is already naturally confined to the thread context. The only additional support needed is about dirty lines in the private caches (in our study the L1 data cache) in the look-ahead mode: They are not used to supply coherence requests from other cores and are not written back on eviction but simply discarded.
- (2) Communication of look-ahead results: In a multicore architecture with shared lower level caches, the look-ahead thread can already warm up the shared caches without any additional support. However, a mechanism to explicitly pass on results from the look-ahead thread is valuable. First, we can send over the branch outcomes. Second, we can also send prefetching hints. This is useful when we want to delay the release of prefetches at the appropriate time, which is helpful for L1 prefetches. We use FIFO buffers for these purposes (Figure 3). Finally, with proper multiplexing support, such buffers can be shared among a

¹In a minority of cases, the configuration with the best performance is not the same as that with the best energy, but when considering the ratio of the performance gain to invested energy, it is still within 5% of the best choice. In other words, if the algorithm were to optimize for performance gain per unit energy invested, it would make largely the same choices and end up only with a slightly different performance and energy profile.

Fig. 3. DLA architectural support.

group of cores so that they can be dynamically configured to pair up any two cores in the group at a given time.

- (3) Support for instruction masking: Finally, we find some convenience in having the code of the look-ahead thread being a subset of the main thread. In other words, the look-ahead thread uses the same program binary and we only manipulate a set of bit masks to indicate whether instructions are to be part of the look-ahead thread or not. With proper hardware support, instructions masked off will be deleted immediately on fetch in the look-ahead thread. These mask bits can be generated either offline or online through dependence analysis of the program binary [31]. We create a few different versions as will be discussed later.

Operations in Detail: When an idle core becomes available, the controller will decide on how to boost the execution speed (Section 4.2). When it chooses to enable the look-ahead mode, the currently executing thread context will be spawned onto the idle core in the look-ahead mode. The bit masks will be stored in a fixed location in the program binary properly aligned and will be fetched into the instruction cache. In other words, when an I-cache miss happens, the controller will issue two read requests to the L2 cache for the instructions (at the address A_i) and their masks stored in address $A_m = f(A_i)$.² Based on the masks embedded in the I-cache, the instruction fetch logic will delete unwanted instructions.

The communication buffers (Figure 3) will be configured accordingly so that the look-ahead thread will write into the queues for the main thread to read from them. Specifically, at the commit time, the outcome of a conditional branch (“taken” or “not taken”) will be stored in FIFO order into a branch outcome queue (BOQ). The main thread reads from the queue at the fetch stage (thus also in program order) and uses the result for branch prediction unless the entry is invalid.³

The BOQ serves a multitude of purposes. First, it passes a branch outcome as a prediction to the main thread. This ensures that in the steady state, the majority of branch mispredictions are experienced only in the look-ahead thread. Second, it is a simple and effective mechanism to detect incorrect look-ahead control flow. When a branch prediction fed by the look-ahead thread turns

²The masks can arrive asynchronously with respect to the actual instructions. Before the actual mask bits arrive, the system defaults to all 1s, thus conservatively including all instructions in the look-ahead thread.

³Both threads will maintain their own head pointers to the queue, advancing the pointers independently. The look-ahead thread does so when writing and the main thread reading. If the main thread has read past the head pointer of the look-ahead thread, then the entries are invalid.

out wrong (which is relatively rare), it means that the look-ahead thread is executing down the wrong path. We will reboot the thread from the current state of the main thread.⁴

Third, we can easily know and control the depth of look-ahead: The number of unread entries in the BOQ equals the number of dynamic basic blocks the look-ahead thread is ahead of the main thread. To prevent run-away prefetching, we only need to limit the size of the BOQ (512 entries in this article). Fourth, it is a convenient way to allow delayed (just-in-time) prefetching. When a prefetch hint is generated, it can be associated with a branch entry and released only on the dequeuing of that BOQ entry.

In addition to the continuous branch direction hints, occasionally the look-ahead thread has other hints such as a data or TLB prefetching hint or a branch target hint. We put these less frequent hints with more information content into another wider but shallower FIFO queue that we call the footnote queue (FQ). When one or more such hints are inserted for a particular basic block, the footnote bit is set for the branch entry in the BOQ telling the main thread to dequeue footnotes. Each footnote entry contains bits to identify the type of hints, the address, and whether this is the last hint for the basic block.

When a memory instruction in the look-ahead thread incurs a primary L1 data cache miss, the controller inserts the cache line address into the FQ. To achieve just-in-time prefetching, we associate this footnote with the branch a few entries upstream. For simplicity, this is done by attaching the footnote to the last entry in BOQ, corresponding to the most recently committed branch instruction. When the main thread acts on the hint and starts the prefetch, it will have some time to overlap with the fetch.

Versions of Skeleton: There are different options in deciding what static instructions to keep on the look-ahead thread. Each option effectively creates a different static code for the look-ahead thread that we call *skeleton*. The most basic version of the skeleton includes all branches and their backward dependence chains and is produced with a binary parser [31]. On top of this, we can add or subtract instructions. Specifically, there are three high-level options to decide:

- L2 prefetch targets (L2): Instructions that account for significant portions of L2 misses can be added to skeleton;
- L1 prefetch targets (L1): Additionally, instructions that account for significant portions of L1 misses can be added to skeleton;
- Biased branches (Br): Conditional branches with a bias over a threshold can be converted to unconditional branches in the skeleton.

Experiments show that the performance benefit of using a particular skeleton, and which set of options are the best depend on the underlying binary. We therefore leave it to the controller to decide on these options.

A combination of these three types of hints can produce up to eight different skeleton versions. For practicality, we use four (Br, Br+L2, L1+L2 and Br+L1+L2) that empirically work well to reduce the search space. The remaining versions are seldom the best performing version in our test cases.

⁴This reboot happens about once every 1 to 10 thousand instructions and the overhead is all modeled in our experiments. When an interrupt, only the main thread needs to be involved in interrupt handling. There are two options to follow. We can reboot the look-ahead thread to execute handler's version of the look-ahead thread if available; or we can suspend reading from the BOQ until the handler returns. If interrupt's frequency is less than or comparable to that of the natural reboot, then the system will not see significant difference.

4.2 Controller

With a number of options to boost performance, the goal of the controller is to find the optimal configuration, i.e., the one that maximizes the benefit (execution time reduction) for the incurred cost (extra energy). The key capability to this goal is to predict the impact of a particular option. There are two components to our proposed solution. First, the controller divides the execution into units. Second, measurements of execution characteristics will be fed into a model that predicts either the impact of various boosting mechanisms or the best option directly.

4.2.1 Dividing Execution into Segments. Prediction of future behavior is always based on the (implicit) assumption that future behavior repeats or at least correlates with the observed past behavior. This assumption is more plausible when the pipeline executes the same code in the period being predicted as in the past period of observation. The quintessential example is a loop. Execution cycles can be broken down into iterations. Statistics collected in one period consisting of one or more iterations can then be used to estimate the same statistics of future iterations. Thus, we propose to break execution sequence into segments corresponding to that of repeating code execution (rather than those with an arbitrary length) and make predictions based on these segments. We discuss an architectural support for doing so, but we note that this can also be done mostly with software using profiling.

There are a number of considerations. First, the units need to be of sufficiently coarse granularity: For short instruction sequences, we can neither accurately measure execution statistics or profitably adjust the system configuration. For this reason, if the inner loops are short in duration, our system ignores them and tries to capture outer loops. Second, the unit needs to capture most of the execution time and yet be simple enough. In some cases, significant execution time is spent in some form of recursion without a canonical looping structure. We thus track function calls in addition to regular backward branches.

The general idea is to capture the “loop branches” that manifest as back-to-back instances of the same backward branch—no other backward branches occur in between. These branches then serve to mark the boundaries between iterations of the same code. Certain complications prevent these branches from occurring back-to-back. For example, a backward branch within the loop body can intersperse among instances of the loop branch. To filter these branches out, we use a loop-branch register to keep track of both the PC and the target of a backward branch. When a newly encountered backward branch matches that stored in the loop-branch register, the loop is identified. We then start to count cycles to measure the granularity of this loop. If we encounter a different backward branch than that in the current loop register, then there are a few different possibilities: The new branch is a loop branch and the branch tracked in the loop branch register is part of the body, the other way around, both are loop branches with one of them being the outer branch, and so on. While the detailed decision logic is shown in Figure 4, the general heuristic is that a backward branch is considered to be nested within another one if its address falls between the PC and the target of the latter. This heuristic does not cover all possibilities but works well for normal code.

Finally, some backward branches are not loop branches. Once the hardware realizes that, it remembers them in a table to help reduce the time it takes to identify a stable loop. In our evaluation we use a Non-Loop PC Table (NLPT) of 20 entries to store the PC of such branches. If a branch is found in this table, then it is skipped by the loop marker.

4.2.2 Characterization of Segments. Once the appropriate loop branches are identified, the execution is divided into segments, each further divided into repeating instances. Depending on the control policy, we may need to characterize the behavior of a segment to decide the most

Cycles	Number of Cycles executed
AluInst	Number of Alu instructions observed
BrInst	Number of branch instructions observed
LdInst	Number of Load instructions observed
StInst	Number of store instructions observed
L1m	L1 Cache Misses encountered
L2m	L2 Cache Misses encountered
Brm	Branch mispredictions encountered

These statistics are used both in training for building models and online when applying the models to pick the right boosting mechanism. In either case, the hardware starts to track the statistics on the detection of a new loop branch in the loop register. On a subsequent execution of the branch where the total number of executed instructions passes the threshold (50,000 instructions in this article), the branch is thus a qualified loop branch and a handler is invoked to process the event.⁵ In offline model building, the handler records the statistics and uses them according to the specific model as will be discussed later. In the case of online application, the handler runs the computation required by the model using the collected statistics and sets the recommended boosting mechanism in the loop table. From this point on, the hardware only needs to apply the selected boosting mechanism and no longer collects statistics for the loop represented by this branch. In our evaluations, we use a loop table of 20 entries, each taking about 8B of storage. Storing more than 20 entries have no noticeable benefits.

Regression Analysis: Regression analysis is perhaps the most commonly applied method to find relationship among variables. We apply regression analysis as follows. In the model building phase,

ACM Transactions on Architecture and Code Optimization, Vol. 15, No. 1, Article 5. Publication date: March 2018.

we run different code segments and measure the effect of a particular boosting mechanism on those segments. We then feed the model with training data where the inputs are the aforementioned execution statistics without any boosting mechanism (Table 1) for every segment and the output is performance gain produced by every boosting configuration.

After training the model, a set of coefficients will be obtained to make a prediction on a new code segment for each of the boosting mechanism. Specifically, the coefficient describes a quadratic function of all statistics:

$$SP_b = \sum_{i,j} \alpha_b(i,j) S_i \times S_j,$$

where SP_b is the predicted speedup of boosting configuration b and S_i 's are the statistics of the code segment. At runtime, on collecting the statistics for a newly discovered segment, the controller evaluates all SP_b and picks the one with the maximum speedup.

The benefit of the regression model is its simplicity. A quick computation of a quadratic formula is all that is needed during runtime. However, this simplicity comes at the expense of simplifying the complex relationship between the behavior of the code and the effectiveness of different boosting mechanisms. This leads to more suboptimal choices as we will see later.

Decision Tree: Another common predictive model is a decision tree. In our case, we provide the execution statistics (Table 1) along with the identity of the boosting configuration that provides the most performance gain for a set of code segments in the offline training phase. We use the Orange data mining package [22] to build the decision tree. At runtime, we feed the execution statistics from a newly encountered code segment to the handler code running the decision. The boosting configuration picked by the decision tree will be used for that code segment.

Compared to the regression model, the decision tree is much more flexible and tends to pick better performing boosting mechanisms more often. The cost of the model is increased online computation to carry out the decision. On average, about 20,000 comparisons are made per decision, roughly 100 times more computation than the regression model. However, since on average a decision is made every 3.9 million instructions, this overhead is still negligible.

Empirical Search: Finally, we also experiment with a straightforward predictive policy of trial and error. There is no training phase involved in this model. Instead, the controller simply goes through all boosting configurations one at a time recording the execution speed (instructions committed per unit time), and selects the configuration showing the most gain.

Compared with the previous two models, the empirical search model requires no offline training but essentially performs an (oversimplified) online training. Unless the number of configurations is very high, this model is practical, and, as we will show later, rather accurate.

4.2.4 Putting It All Together. As explained in the beginning of this section, the goal of this controller is to identify the optimal configuration for each segment of the program. To that effect, loop hardware identifies loop boundaries to detect segments, and prediction models find the best configuration for that segment. Figure 5 shows a descriptive diagram of the workings of ATB using empirical search model. At every branch commit, loop hardware is consulted to recognize if this particular instruction marks the beginning a new loop or a new iteration of the same loop. The empirical search model requires only one of the eight statistics described in Table 1 (Cycles). At the beginning of every new loop, the model checks if the optimum configuration for this loop is already present in the Loop Control Table (LCT). If found, then ATB informs the boosting mechanism of the optimum configuration and no further action is necessary while this loop is being executed. If LCT does not contain any information about the new loop, then Loop Register (LR) is reset and

Fig. 5. Runtime support for empirical search controller.

LoopBr field is updated with PC of the loop branch.⁶ The empirical search model enables every boosting configuration for a few (*Loop Threshold*) loop iterations and identifies the configuration that provides maximum speedup (IPC). Once the optimum boosting configuration is identified, its information is stored in LCT and the system is informed of the right boosting mechanism.

Note that the controller is only invoked when the loop hardware identifies a new loop branch. There are times when a program executes code that does not belong to a discernible loop before transitioning to another loop. In our evaluations such transient phases make up for less than 2% of the execution time. To reduce unnecessary mode switching overheads, during transients, the system continues to the configuration set for the previous loop.

5 EXPERIMENTAL ANALYSIS

Using execution-driven simulations (Section 5.1), we first show (in Section 5.2) the primary figures of merit for the proposed system. We show that significantly more performance gain is obtained in the new boosting design than adjusting clock speed only. We then perform an in-depth analysis of various components and usage scenarios of the design (Section 5.2 to 5.4).

5.1 Simulation Setup

We model various configurations using execution-driven simulation. Our simulator is based on Gem5 [13]. The baseline system is a multicore chip with out-of-order cores. Table 2 lists additional technical details.

The turbo boosting configurations are loosely based on an Intel i7 chip [40], where a baseline frequency of 3GHz (at 0.8V) can be scaled up to seven steps, each step being 133.3MHz [39], and a corresponding voltage increase of 50mV [17]. For DLA we model a communication delay of three cycles between lead core and main core, similar to an L1 cache access. For energy consumption modeling, we use McPAT [45] and assume a 22nm technology node. We modified McPAT to correctly model our proposed architecture. All the pipeline stages, buffers, caches, network and memory are faithfully modeled. The baseline configuration operates at 3GHz and all the figures are normalized to it unless otherwise mentioned. We evaluate our proposal on multiple benchmark suites. In addition to SPEC2006 [35] we use CRONO [1] (graph applications) and STARBENCH [3] (embedded applications) benchmark suites. For SPEC2006 we use reference inputs. For STARBENCH

⁶For decision tree and regression analysis, all the statistics are collected for a few loop iterations (denoted by *Loop Threshold*) during which the processor runs at base operating frequency with no DLA support. When sufficient loop iterations have executed, a prediction is made and boosting mechanism is informed accordingly.

Table 2. Processor Configuration

Processor	4 cores
Operating Points	Base: 0.8V, 3.0GHz Turbo Boost: 1.15V, 4.0GHz
Core	7-stage pipeline, out-of-order, 4-wide, 192 ROB, 96 LSQ, 128INT/128FP PRF, 4INT/ 2MEM/ 4FP FUs
L1 Caches	Split I/D, 64KB, 4-way, 64B blocks, 3ports, 1ns hit latency, 32 MSHRs
L2 Cache	2MB, 8-way, 64B blocks, 1 port, 10ns hit latency
Prefetcher	Stride Prefetcher, 64 entry
Main Memory	4GB, DDR3 1600MHz, 1.5V, 2 channels, 2 ranks/channel, 8 banks/rank, $t_{RCD}=13.75ns$, $t_{RAS}=35ns$, $t_{FAW}=30ns$, $t_{WTR}=7.5ns$, $t_{RP}=13.75ns$
DLA	512 Entry BOQ (128 bytes, 1 Read/Write port, 1ns access latency), 128 Entry FootNote Queue (1kB, 1 Read/Write port, 1ns access latency)
ATB	Boost States: 5 (Frequency Boosting and 4 DLA Skeletons), LR (22 bytes), 20 Entry LCT (90 bytes), LH (90 bytes)

Fig. 6. Speedup for SPEC (s.xxx), Crono (c.xxx), and sTarbench (t.xxx) benchmarks under frequency-only boosting (FB), decoupled look-ahead (DLA), and adaptive turbo boosting (ATB), which uses the empirical model for controller. The ATB bar is divided into a shaded portion and an empty portion. The shaded portion shows gain obtained by ATB without using any frequency boosting. The empty portion represents the additional performance gain when it has the option of frequency boosting.

we picked large inputs from the provided input set and for CRONO we use graph input data structures from google, amazon, twitter mathoverflow and california road-networks. All benchmarks are compiled using gcc with -O3 option. We use simpoint methodology to accurately capture all phases of the application. We generate five simpoints per benchmark with 10 million instruction interval. Simpoints are generated using the SimPoint Tool [62]. Note that simulation results are prone to errors due to simulators [6], sampling methodology [62], baseline configuration, and so on. We report relative numbers instead of absolute values in an effort to reduce the impact of such errors.

5.2 Primary Figures of Merit

Performance Benefit: A primary use of today's turbo boosting mechanism is to leverage the power and thermal budget made available by one or more idle cores in a multicore system. Our main argument for this article is that in those situations, the best strategy to leverage the idle core depends on characteristics of the application, and thus an adaptive turbo boosting architecture is needed. With proper support for ATB, we can improve performance much more than that achievable with just conventional frequency-only boosting. Figure 6 shows the speedup of applications under three different boosting policies: frequency-only boosting (FB), always employing DLA, and ATB. In this figure, we only show one variant that uses the empirical search controller. For visual

Fig. 7. Execution time distribution between FB and DLA in ATB.

clarity, we sort the applications based on increasing ratio of $\frac{SP_{ATB}}{SP_{FB}}$, where SP_{ATB} and SP_{FB} are the speedups of ATB and FB, respectively.

From the figure, we can see three main things.

- (1) As already suggested early in Figure 1, boosting clock frequency and performing look-ahead have complementarity: When one shows a low performance gain, the other tends to show a high gain. Not surprisingly therefore, having the two options at our disposal is much better: When FB does not work well for a particular code segment, ATB will choose to deploy DLA and vice versa. Even though the major benefit of ATB is from the complementarity between DLA and FB, we note that ATB allows a better implementation of DLA. Through tuning, ATB can pick a better skeleton compared to the heuristically derived skeleton. This effect can be seen in the shaded portion of the ATB: Without frequency boosting, ATB can improve DLA's overall performance by 4.8%. The inclusion of frequency boosting provides an additional performance gain of 16%.
- (2) With traditional FB, the speedup ranges from 1.0 to 1.5 with a (geometric) mean of 1.19. With our ATB, the speedup ranges from 1.05 to 2.34 with a mean of 1.4, essentially doubling the gain in performance on average. With the exception of the first five applications, the benefit of ATB over FB ranges from noticeable to dramatic. Figure 7 shows the resulting distribution of execution between the two boosting mode under the ATB controller.
- (3) For the Crono benchmark suite, the benefit of DLA and ATB is more pronounced than that in the SPEC suite, whereas the benefit of increasing frequency is less. As our computing systems are increasingly used towards the analysis of large data sets, the case for including support for DLA is more compelling.

It is not always possible to use the frequency boosting feature at its maximum potential. Runtime factors like core temperature, estimated current/power consumption, the number of active cores and so on, determine the amount of boost that can be supported by the processor. Figure 8 shows the impact of FB and ATB under different boosting headrooms. Note that ATB is even more appealing when the range of frequency boosting is limited. As the frequency boost reduces from 1GHz down to 330MHz, the amount of performance improvement obtained by FB drops from 19% eventually to 9%. Comparatively, ATB's benefit lowers from 40% to 34% at most. It is worth mentioning that with limited frequency boosting, ATB favors DLA under certain epochs that earlier favored frequency boosting.

Energy Overhead: Another consideration for a turbo boosting mechanism is the energy cost paid for the performance gain. In this respect, our proposed system is also an improvement over the conventional frequency boosting. Before we inspect experimental data, we first look at a common approximation that energy cost doubles in DLA due to executing the program twice. Though a

Fig. 8. Speedup under limited frequency boosting opportunities.

Fig. 9. Normalized energy consumption under different boosting strategies. Shaded portion represents the static energy component.

tempting assumption, it is nevertheless a *significant* overestimation. First, not all instructions are on the look-ahead thread. On average, 66% of instructions are on the look-ahead thread even if the system always runs in DLA mode. In ATB, DLA is enabled for about 62% of time, leading to roughly 44% extra committed instructions.

Second, adding these 44% instruction does not increase energy overhead by the same proportion as many activities are not duplicated but only time shifted. For instances, the baseline microarchitecture decodes 25% more instructions than committed. In DLA, wrong-path instructions are almost completely avoided in the main thread. Also memory accesses are mostly just time shifted, with the traffic being only 3% higher in baseline DLA than the underlying microarchitecture.

Third, still other overheads do not increase at all or even decrease. Leakage and clock distribution energy for the main thread generally decreases due to shorter execution time.

Roughly speaking, about 52% of baseline energy budget increases proportional to added instructions; 12% remains constant regardless of added instructions; and 36% decreases proportional to cycles saved. Combining these contributions together, Figure 9 shows the energy consumption of the whole system under different boosting strategies.

We can see two things from the figure. First, the energy cost of boosting tends to have a negative correlation with performance, because reduction in execution time directly translates to reduced static energy cost. As a result, the energy overhead is high when the particular boosting mechanism has a small performance benefit. This leads to the second observation that since our ATB naturally avoids the ineffective configurations, it also avoids the high energy overhead. Take *s.xa* for instance, ATB avoids the high energy overhead of DLA. On average, the energy overhead of ATB, at 22%, is about half of that in frequency-only boosting (45%).

Overall, with half of the energy cost and twice the performance gain, ATB is not only decidedly superior to FB, it actually lowers the system's energy-delay product on average.

Fig. 10. (a) Prediction errors from different models. The errors for each segment is the relative execution time difference between the configuration chosen by the model and the actual best-performance configuration. Each error is further divided into model error and sampling error. For each application, the overall error is the weighted average of all segments. For clarity, only the suite average is shown together with a subset of the applications' results. (b) The speedup obtained with each controller for the same applications.

5.3 Controller Analysis

Prediction Errors: For ATB, the controller needs to make good predictions about the effects of different configurations to make a judicious choice. Error in predictions lead to suboptimal choices. We quantify such error from the three prediction models (Section 4.2.3).

Recall that in all three models, the system obtains execution parameters for a period of time (a few iterations of a detected segment S_i) and makes a prediction of the best boosting configuration C_k . We ran simulations to get the actual best execution time for S_i among all configurations: $T_{min} = \min_j T(S_i, C_j)$. The difference between the time of the predicted best configuration and T_{min} is considered the error. Thus, the relative prediction error for segment S_i is $\frac{T(S_i, C_k) - T_{min}}{T_{min}}$, and the error for the entire application is a weighted average of those for each segment. We show this error for all three prediction models and applications in Figure 10.

To understand this error, we loosely break it down into two components due to the two assumptions made in the prediction process:

- (1) Given the execution statistics, the model can accurately predict the effect of all boosting configurations; and
- (2) The observed behavior in the initial sample iterations accurately represents the entire segment.

The error due to the first assumption (which we call the model error) can be quantified by feeding the model with the exact execution statistics of the segment *post hoc*. Note that for the empirical model, there is no model error by definition. The remaining error (which we call sampling error) can be attributed to assumption 2.

From Figure 10(a), we compare the errors of the three different prediction models. For clarity, we only show a subset of the applications from SPEC2006 benchmark suite. Specifically, we sort all applications by the sum of errors from all three models and show every other application. The general observation is the same across all applications: First, for the same application, higher prediction errors correspond to more suboptimal choices and thus lower overall speedup. Second, the sampling error across all three models are reasonably close to each other (on average, about 3.2–3.5%). The decision tree model, being more sophisticated than regression, has a lower model error (2.1% vs 3.7%) and a better performance in general (a speedup of 1.33 vs. 1.32). The empirical approach essentially forgoes the prediction model altogether and thus has no model error and better performance (speedup of 1.37) than the other two.

Fig. 11. Comparison of an average relative deviation of statistics between segments discovered by our segment-marking hardware and between neighboring segments of a fixed length.

Segment Marking: In our proposed design, we rely on architectural support to mark segments so that they are based on the same code executed. This is to reduce the sampling error. To justify the architectural support, we compare to an often-used simpler alternative that assumes behavior stays stable within a “phase” and goes through phase changes with observable effects on execution statistics.

In this traditional approach, for each fixed-length segment (100,000 instructions in this experiment), we collect execution statistics to act as a “signature.” Any significant deviation of the signature in a new segment suggests the start of a new phase. At the beginning of new phase, statistics collected for a fixed-length segment will be assumed to represent the behavior of the phase and drive the boosting configuration decision.

Figure 11 contrasts these two approaches. A direct comparison is to measure the performance of the system under two controllers where the only difference is how execution is broken into self-similar units. In this experiment, we use the regression prediction model and show the difference in speedup in Figure 11(a). We see that in general as well as on average, using our proposed segment marking hardware produces noticeably better results. This is largely because by better capturing the repeating code structure, the assumption of predictable future behavior holds better. Figure 11(b) quantifies this by measuring the deviation of statistics of the segments.

Specifically, in both cases, statistics (Table 1) are measured for a segment sm and are used to predict those for another segment sp . The deviation for statistic St is thus the absolute different between their respective values St_{sm} and St_{sp} . Since these statistics can have different magnitudes, we normalize the deviations to average them meaningfully. To do so, we take the global mean value of the statistic $mean(St)$ as the standard yard stick for normalization. For example, if the statistic is branch mispredicts, then the mean value would be the global average mispredicts per 100,000 instructions across all applications. Note that the exact value of the mean is not nearly as important as its order of magnitude. We then average the normalized deviation ($\frac{|St_{sm}-St_{sp}|}{mean(St)}$) across all statistics, and then weighted-average across all segments. Figure 11(b) compares the per-application average deviation between the two approaches. We can see that the average deviation is much lower when using our segment marking hardware, indicating better similarity of behavior between segments deemed similar.

5.4 Other Usage Scenarios

In Section 5.2 we showed that with DLA support and an adaptive controller, ATB can achieve much more significant performance benefit than FB for less energy consumption, thanks to the fact that DLA is often the more effective boosting mechanism. But the benefit of DLA support goes beyond that, as we show in these following usage scenarios.

Fig. 12. The speedup (a) and normalized energy (b) of maximum boosting for SPEC benchmarks. In this scenario, the speedup obtained is between 1.18 and 2.5 with a geometric mean of 1.55. Panel (c) shows the equivalent operating frequency to achieve the same speedup.

Boosting Range: In the earlier experiment, the controller chooses *either* DLA or FB, not both. This has the side effect of limiting the power consumption to well within twice the baseline power consumption. When the thermal and power headrooms allow, the controller can choose both and thus materially extends the range of boosting achievable. In the following experiment, we select the best-performing option without concern for the overall power.

Figure 12(a) shows the speedup obtained for SPEC benchmarks. We can see that the range of boosting is further expanded to up to 2.5x with a geometric mean of 1.55x compared with 1.37x before. The energy cost rises to 1.48x (Figure 12(b)). Note that this is a non-trivial extension of the boosting range. To achieve the same effect with only increasing clock frequency would require the processor to raise the frequency by 100% on average to more than 6GHz and sometimes even more (Figure 12(c)). At these levels, it is no longer just a matter of paying exorbitant energy costs, but the feasibility.

Broader Choices: For high-performance processors, frequency boost usually has a very limited range [40], and a resulting limited range of speedups. Decoupled look-ahead, on the other hand, has a much wider range of achievable speedups. Consequently, in an multi-program environment with a variety of threads, there is more chance to find a target thread that can receive significant speedup.

To illustrate this point more concretely, consider a 4-core processor chip with three active threads and one idle core that provides the turbo boosting. To exploit the additional freedom of selecting which thread to boost, the controller will be extended as follows. When any thread enters a new segment, the controller calculates the benefit of boosting the new segment and compares that to the current target of boosting. If the benefit is sufficiently higher, then we can terminate the

Table 3. Description of Mixes

m1 : t.rr, c.co, s.bz	m2 : s.li, s.om, s.bw	m3 : t.rc, c.tc, s.sp	m4 : s.h2, s.gr, s.le
m5 : s.go, s.hm, s.mc	m6 : c.cc, c.co, t.md	m7 : s.as, s.lb, s.po	m8 : t.md, s.mc, t.rt
m9 : c.ap, s.mc, s.mi	m10 : c.dfs, c.bc, s.gr	m11 : t.rr, c.ap, s.lb	m12 : t.st, t.rt, t.ti

Fig. 13. Speedup of multi-program version of FB and ATB in a four-core system with one idle core available to allow turbo boosting. The geometric mean of the effective speedup of MP-FB is 1.23 and the effective speedup MP-ATB is 1.57.

boosting for the current target and engage boosting for the new segment. Of course, a practical implementation would have to consider a number of secondary issues.⁷

We model this scenario with randomly chosen mixes of three applications each (Table 3). We use the empirical controller modified as discussed above. In this experiment, we favor the thread that will receive the most speedup (relative increase of speed) as the target of boosting.⁸

In Figure 13, we show the result of the multi-program (MP) version of both FB and ATB. Since the controller can change the target of boosting, the three programs can each gain some part-time boosting. In order to meaningfully compare with the earlier result where boosting is applied to a fixed thread (Figure 6), we piece together the portion of the program being boosted and treat it as a virtual program with its own baseline execution time. Effective speedup is the speedup of this virtual program. As we can see, MP-FB averages 1.23 effective speedup for these mixes, which is a small improvement over the speedup of 1.19 for FB in the single-program case. In contrast, MP-ATB averages 1.57 effective speedup, which is a more significant improvement over 1.4x for ATB.

The key to this improvement is that we can choose to “invest” boosting to the threads with the best return. Based on the characteristics of frequency and look-ahead boosting, the effect is statistically much more pronounced for look-ahead. To show this point more quantitatively, we perform a statistical analysis, taking thousands of random mixes of applications and given the speedup observed for each segment under different boosting, we can estimate the effective speedup obtained with MP-FB and MP-ATB. Without using execution driven simulation, this estimate ignores some inefficiencies and is thus a bit optimistic: For the mixes shown in Figure 13, the statistical estimate of effective speedup is 1.22 for MP-FB and 1.56 for MP-ATB. In other words, the error with the statistical estimate appears to be negligible.

With the statistical analysis we can obtain the probability distribution of the effective speedup for both MP-FB and MP-ATB. We present this probability distribution in a form that shows the probability of obtaining a particular speedup or better. Figure 14 summarizes the implications some of which already discussed:

⁷For instance, we may want to take into account the average length of a segment to ensure the benefit of switching justifies the overhead.

⁸We have also experimented with the case when the user would prefer to increase system throughput. In that case, the controller selects the thread that would enjoy the most increase in throughput (or absolute increase in speed). The broad conclusions are the same.

Fig. 14. A modified cumulative distribution function for the effective speedup of MP-FB and MP-ATB. The y value of a point on the curve shows the probability of obtaining a speedup greater than or equal to the x value of the point. The higher the curve the better. Panels (a) and (b) show the result when the controller can choose among three or seven applications to boost, respectively.

- (1) The MP-ATB curves move significantly to the right (or upside) of that of MP-FB. The opening between the two curves summarizes the benefit for including decoupled look-ahead support: For a typical situation, it is likely to significantly improve the speedup, or, alternatively, there is more chance to achieve the same target speedup.
- (2) The expectation of the speedup of MP-ATB (1.56) is significantly higher than that of MP-FB (1.22).
- (3) The range of achievable speedup is greatly extended. In a minority of cases, the effective speedup of MP-ATB goes beyond 2, suggesting that in those situations, it may even be more profitable to intentionally vacate a core for boosting use.

6 CONCLUSIONS

When operating conditions allow, processor chips can choose to invest more energy to speed up execution in a turbo boosting mode. Today's turbo boosting mechanisms only involve increasing clock frequency at a higher operating voltage. In this article, we have argued for an ATB architecture that is significantly more effective in achieving performance gain and at the same time more energy efficient than an FB design. To enable ATB, we need to add the support for decoupled look-ahead execution and runtime support to make judicious decision about the right boosting actions for the executing code.

Through experimental analysis, we have shown that, thanks to the complementarity of decoupled look-ahead and frequency boosting, ATB is much more effective at improving performance. When keeping the power roughly the same, it achieves a 1.4 average speedup compared to 1.19 with FB, while halving the energy overhead from 45% to 22%. ATB also greatly extends the range of achievable speedup, especially in a multi-program environment where it can choose among multiple target threads to accelerate. In a number of randomly created application mixes, ATB can achieve an even higher effective speedup of 1.56. Overall, compared to existing turbo boosting designs, the proposed adaptive turbo-boosting architecture is a much more effective and yet more energy-efficient solution.

REFERENCES

- [1] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan. 2015. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 44–55.
- [2] D. Albonesi, R. Balasubramonian, S. Dropsho, S. Dwarkadas, M. Huang, V. Kursun, G. Magklis, M. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. Cook, and S. Schuster. 2003. Dynamically tuning processor resources with adaptive processing. *IEEE Comput.* 36, 12 (Dec. 2003), 49–58.
- [3] M. Andersch, B. Juurlink, and C. Chi. 2013. A benchmark suite for evaluating parallel programming models. In *Proceedings of the Workshop on Parallel Systems and Algorithms (PARS'13)*, Vol. 28. 1–6.

- [4] M. Annavaram, J. Patel, and E. Davidson. 2001. Data prefetching by dependence graph precomputation. In *Proceedings of the International Symposium on Computer Architecture*. 52–61.
- [5] A. Ansari, S. Feng, S. Gupta, J. Torrellas, and S. Mahlke. 2013. Illusionist: Transforming lightweight cores into aggressive cores on demand. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA '13)*, 2013.
- [6] G. Anthony, J. Pusdesris, R. Dreslinski, T. Mudge, C. Sudanthi, C. Emmons, M. Hayenga, and N. Paver. 2014. Sources of error in full-system simulation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*. 13–22.
- [7] I. Atta, X. Tong, V. Srinivasan, I. Baldini, and A. Moshovos. 2015. Self-contained, accurate precomputation prefetching. In *Proceedings of the International Symposium on Microarchitecture*. 153–165.
- [8] T. Austin. 1999. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the International Symposium on Microarchitecture*. 196–207.
- [9] T. Austin, D. Blaauw, T. Mudge, and K. Flautner. 2004. Making typical silicon matter with razor. *IEEE Comput.* 37, 3 (Mar. 2004), 57–65.
- [10] R. Barnes, E. Nystrom, M. Merten, and W. Hwu. 2002. Vacuum packing: Extracting hardware-detected program phases for post-link optimization. In *Proceedings of the International Symposium on Microarchitecture*. 233–244.
- [11] R. Barnes, E. Nystrom, J. Sias, S. Patel, N. Navarro, and W. Hwu. 2003. Beating in-order stalls with “Flea-flicker” Two-pass pipelining. In *Proceedings of the International Symposium on Microarchitecture*. 387–399.
- [12] A. Batson and A. Madison. 1976. Measurements of major locality phases in symbolic reference strings. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 75–84.
- [13] N. Binkert et al. 2011. The gem5 simulator. *ACM SIGARCH Comput. Arch. News* 39, 2 (2011), 1–7.
- [14] S. Campanoni, K. Brownell, S. Kanev, T. Jones, G. Wei, and D. Brooks. 2014. HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs. In *Proceedings of the International Symposium on Computer Architecture*. 217–228.
- [15] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout. 2015. The load slice core microarchitecture. In *Proceedings of the International Symposium on Computer Architecture*. 272–284.
- [16] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. 1999. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the International Symposium on Computer Architecture*. 186–195.
- [17] J. Charles, P. Jassi, A. Narayan S, A. Sadat, and A. Fedorova. 2009. Evaluation of the intel core i7 turbo boost feature. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 188–197.
- [18] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. 2005. High-performance throughput computing. *IEEE Micro* 25, 3 (May/June 2005), 32–45.
- [19] D. Chaver, L. Pinuel, M. Prieto, F. Tirado, and M. Huang. 2003. Branch prediction on demand: An energy-efficient solution. In *Proceedings of the International Symposium on Low-Power Electronics and Design*. 390–395.
- [20] R. Cochran, C. Hankendi, A. Kivilcim Coskun, and S. Reda. 2011. Pack and cap: Adaptive DVFS and thread packing under power caps. In *Proceedings of the International Symposium on Microarchitecture*.
- [21] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. 2001. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the International Symposium on Computer Architecture*. 14–25.
- [22] T. Curk, J. Demsar, Q. Xu, G. Leban, U. Petrovic, I. Bratko, G. Shaulsky, and B. Zupan. 2005. Microarray data mining with visual programming. In *Bioinformatics*.
- [23] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. 2012. CoScale: Coordinating CPU and memory system DVFS in server systems. In *Proceedings of the International Symposium on Microarchitecture*. 143–154.
- [24] A. Dhodapkar and J. Smith. 2003. Comparing program phase detection techniques. In *Proceedings of the International Symposium on Microarchitecture*. 217–227.
- [25] M. Dubois and Y. Song. 1998. Assisted execution. Technical Report, Department of Electrical Engineering, University of Southern California.
- [26] E. Duesterwald, C. Cascaval, and S. Dwarkadas. 2003. Characterizing and predicting program behavior and its variability. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 220–231.
- [27] J. Dundas and T. Mudge. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the International Conference on Supercomputing*. 68–75.
- [28] H. Esmailzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley. 2011. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 319–332.
- [29] S. Fan, S. Zahedi, and B. Lee. 2016. The computational sprinting game. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 561–575.
- [30] A. Farcy, O. Temam, R. Espasa, and T. Juan. 1998. Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes. In *Proceedings of the International Symposium on Microarchitecture*. 59–68.

- [31] A. Garg and M. Huang. 2008. A performance-correctness explicitly decoupled architecture. In *Proceedings of the International Symposium on Microarchitecture*. 306–317.
- [32] A. Garg, R. Parihar, and M. Huang. 2011. Speculative parallelization in decoupled look-ahead. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 412–422.
- [33] B. Greskamp and J. Torrellas. 2007. Paceline: Improving single-thread performance in nanoscale CMPs through core overclocking. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 213–224.
- [34] M. Hashemi, O. Mutlu, and Y. Patt. 2016. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’16)*. 1–12.
- [35] J. L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Comput. Arch. News* 34, 4 (Set. 2006), 1–17.
- [36] A. Hilton, N. Eswaran, and A. Roth. 2009. CPROB: Checkpoint processing with opportunistic minimal recovery. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*. 159–168.
- [37] C. H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski. 2015. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 271–282.
- [38] M. Huang, D. Chaver, L. Pinuel, M. Prieto, and F. Tirado. 2003. Customizing the branch predictor to reduce complexity and energy consumption. *IEEE Micro* 23, 5 (Sep. 2003), 12–25.
- [39] Intel Corporation. 2008. *Intel Corporation. Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors*. November 2008.
- [40] Intel Corporation. 2015. *Intel Core i7-900 Desktop Processor Extreme Edition Series and Intel Core i7-900 Desktop Processor Series*. 2015.
- [41] S. Nek kalapu, H. Akkary, K. Jothi, R. Retnamma, and X. Song. 2008. A simple latency tolerant processor. In *Proceedings of the IEEE International Conference on Computer Design*. 384–389.
- [42] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt. 2012. MorphCore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP. In *Proceedings of the International Symposium on Microarchitecture*. 305–316.
- [43] D. Kim and D. Yeung. 2004. A study of source-level compiler algorithms for automatic construction of pre-execution code. *ACM Trans. Comput. Syst.* 22, 3 (2004), 326–379.
- [44] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. 2002. A large, fast instruction window for tolerating cache misses. In *Proceedings of the International Symposium on Computer Architecture*. 59–70.
- [45] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009. McPAT: An integrated power, area and timing modeling framework for multicore and manycore architectures. In *Proceedings of the International Symposium on Microarchitecture*.
- [46] D. Lo and C. Kozyrakis. 2014. Dynamic management of turbomode in modern multi-core chips. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 603–613.
- [47] C. Luk. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the International Symposium on Computer Architecture*. 40–51.
- [48] C. Madriles, P. López, J. Codina, E. Gibert, F. Latorre, A. Martinez, R. Martinez, and A. Gonzalez. 2009. Boosting single-thread performance in multi-core systems through fine-grain multi-threading. In *Proceedings of the International Symposium on Computer Architecture*. 474–483.
- [49] M. Merten, A. Trick, C. George, J. Gyllenhaal, and W. Hwu. 1999. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proceedings of the International Symposium on Computer Architecture*. 136–147.
- [50] F. Mesa-Martinez and J. Renau. 2007. Effective optimistic-checker tandem core design through architectural pruning. In *Proceedings of the International Symposium on Microarchitecture*. 236–248.
- [51] R. Miftakhutdinov, E. Ebrahimi, and Y. N. Patt. 2012. Predicting performance impact of DVFS for realistic memory systems. In *Proceedings of the International Symposium on Microarchitecture*. 155–165.
- [52] A. Moshovos, D. Pnevmatikatos, and A. Baniassadi. 2001. Slice-processors: An implementation of operation-based prediction. In *Proceedings of the International Conference on Supercomputing*. 321–334.
- [53] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 129–140.
- [54] S. Palacharla, N. Jouppi, and J. Smith. 1997. Complexity-effective superscalar processors. In *Proceedings of the International Symposium on Computer Architecture*. 206–218.

- [55] R. Parihar and M. Huang. 2014. Accelerating decoupled look-ahead via weak dependence removal: A metaheuristic approach. In *Proceedings of the International Symposium on High-Performance Computer Architecture*.
- [56] I. Paul, S. Manne, M. Arora, W. L. Bircher, and S. Yalamanchili. 2013. Cooperative boosting: Needy versus greedy power management. In *Proceedings of the International Symposium on Computer Architecture*. 285–296.
- [57] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. 2000. A study of slipstream processors. In *Proceedings of the International Symposium on Microarchitecture*. 269–280.
- [58] A. Raghavan, L. Emurian, L. Shao, M. C. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. K. Martin. 2013. Computational sprinting on a hardware/software testbed. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 155–166.
- [59] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. K. Martin. 2012. Computational sprinting. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 1–12.
- [60] A. Roth and G. Sohi. 2001. Speculative data-driven multithreading. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 37–48.
- [61] X. Shen, Y. Zhong, and C. Ding. 2004. Locality phase prediction. In *Proceedings of the International Conference on Arch. Support for Prog. Lang. and Operating Systems*.
- [62] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 45–57.
- [63] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. 2004. Discovering and exploiting program phases. *IEEE Micro* 23, 6 Nov.-Dec. 2004), 84–93.
- [64] T. Sherwood, S. Sair, and B. Calder. 2003. Phase tracking and prediction. In *Proceedings of the International Symposium on Computer Architecture*. 336–347.
- [65] R. Shioya, M. Goshima, and H. Ando. 2014. Front-end execution architecture for high energy efficiency. In *Proceedings of the International Symposium on Microarchitecture*. 419–431.
- [66] B. Sinharoy, J. Van Norstrand, R. Eickemeyer, H. Le, J. Leenstra, D. Nguyen, B. Konigsburg, K. Ward, M. Brown, J. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. Fernsler. 2015. IBM POWER8 processor core microarchitecture. *IBM J. Res. Dev.* 59, 1, 2.
- [67] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. 2004. Continual flow pipelines. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [68] B. Su, J. Gu, L. Shen, W. Huang, J. L. Greathouse, and Z. Wang. 2014. PPEP: Online performance, power, and energy prediction framework and DVFS space exploration. In *Proceedings of the International Symposium on Microarchitecture* 445–457.
- [69] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. 2000. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 257–268.
- [70] A. Vega, A. Buyuktosunoglu, H. Hanson, P. Bose, and S. Ramani. 2013. Crank it up or dial it down: Coordinated multiprocessor frequency and folding control. In *Proceedings of the International Symposium on Microarchitecture*. 210–221.
- [71] Y. Yao and Z. Lu. 2016. DVFS for NoCs in CMPs: A thread voting approach. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 309–320.
- [72] H. Zhou. 2005. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 231–242.
- [73] C. Zilles and G. Sohi. 2001. Execution-based prediction using speculative slices. In *Proceedings of the International Symposium on Computer Architecture*. 2–13.
- [74] C. Zilles and G. Sohi. 2002. Master/slave speculative parallelization. In *Proceedings of the International Symposium on Microarchitecture*. 85–96.
- [75] S. Pugsley, Z. Chishti, C. Wilkerson, P. Chuang, R. Scott, A. Jaleel, S. Lu, K. Chow, and R. Balasubramonian. 2014. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 626–637.
- [76] J. Kim, E. Teran, P. Gratz, D. Jiménez, S. Pugsley, and C. Wilkerson. 2017. Kill the program counter: Reconstructing program behavior in the processor cache hierarchy. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 737–749.
- [77] S. Kondguli and M. Huang. 2017. T2: A highly accurate and energy efficient stride prefetcher. In *Proceedings of the International Conference of Computer Design*. 373–376.

Received May 2017; revised October 2017; accepted November 2017