# Illusionist: Transforming Lightweight Cores into Aggressive Cores on Demand

Amin Ansari
University of Illinois
amina@illinois.edu

Shuguang Feng
Northrop Grumman Corp.
shoe@umich.edu

Shantanu Gupta
Intel Corp.
shantanu.g.gupta@intel.com

Josep Torrellas
University of Illinois
torrella@illinois.edu

Scott Mahlke
University of Michigan
mahlke@umich.edu

## Abstract

*Power dissipation limits combined with increased silicon integration have led microprocessor vendors to design chip multiprocessors (CMPs) with relatively simple (lightweight) cores. While these designs provide high throughput, single-thread performance has stagnated or even worsened. Asymmetric CMPs offer some relief by providing a small number of high-performance (aggressive) cores that can accelerate specific threads. However, threads are only accelerated when they can be mapped to an aggressive core, which are restricted in number due to power and thermal budgets of the chip. Rather than using the aggressive cores to accelerate threads, this paper argues that the aggressive cores can have a multiplicative impact on single-thread performance by accelerating a large number of lightweight cores and providing an* illusion *of a chip full of aggressive cores. Specifically, we propose an adaptive asymmetric CMP, Illusionist, that can dynamically boost the system throughput and get a higher single-thread performance across the chip. To accelerate the performance of many lightweight cores, those few aggressive cores run all the threads that are running on the lightweight cores and generate execution hints. These hints are then used to accelerate the execution of the lightweight cores. However, the hardware resources of the aggressive core are not large enough to allow the simultaneous execution of a large number of threads. To overcome this hurdle, Illusionist performs aggressive dynamic program distillation to execute small, critical segments of each lightweight-core thread. A combination of dynamic code removal and phase-based pruning distill programs to a tiny fraction of their original contents. Experiments demonstrate that Illusionist achieves 35% higher single thread performance for all the threads running on the system, compared to a CMP with all lightweight cores, while achieving almost 2X higher system throughput compared to a CMP with all aggressive cores.*

## 1 Introduction

Building high-performance, monolithic microprocessors has become impractical due to their excessive power dissipation, thermal issues, design complexity, and early stage failures [28, 17]. As a result, there has been a major shift in the industry towards chip multiprocessors (CMPs) [23, 16, 6, 22]. These CMPs with multiple processing cores rely on different sources of parallelism to extract the most throughput from the threads that can be run on the system. As the number of cores increases, to deal with the power envelope, the complexity of these cores needs to decrease. By integrating a large number of these relatively simple cores (i.e., lightweight cores) on the same die, the single-thread performance suffers for most applications compared to traditional monolithic processors such as Intel Core i7. This results in a poor user experience and in certain scenarios missing the timing deadlines of real-time applications. To mitigate this problem, asymmetric chip multiprocessors (ACMPs) have been introduced in the past [10, 19, 20] which effectively execute parts of a program that cannot be parallelized effectively. Most ACMPs have a large number of lightweight cores and a few aggressive cores. In such a system, the scheduler tries to assign jobs whose performances are more critical to the system to these aggressive cores.

While ACMPs provide some improvement, such designs are not a panacea for two reasons. First, the number of aggressive cores is restricted to a small number to ensure the chip is within its power and thermal budgets. As a result, the number of threads that can be simultaneously accelerated is also restricted by the availability of aggressive cores. Second, the number of aggressive cores is fixed at design time thereby providing no adaptability to the single-thread performance requirements of individual workloads. Ideally, the number of aggressive cores could be configured by the operating system to adapt to the workload demands.

In this paper, we postulate a counter-intuitive argument: *To maximize single-thread performance, aggressive cores should not be used to accelerate individual threads, but rather should redundantly execute threads running on the lightweight cores.* On the surface, this sounds like a waste of resources, but as will be shown, the result is a multiplicative increase in single-thread performance of the entire chip by accelerating the performance of large groups of threads. Multiplicative single-thread performance is part of the broader objective of *Illusionist* and is illustrated in a simple way in Figure 1. As shown, we start from a CMP system at the top that has N cores, 12 in this example. Illusionist's

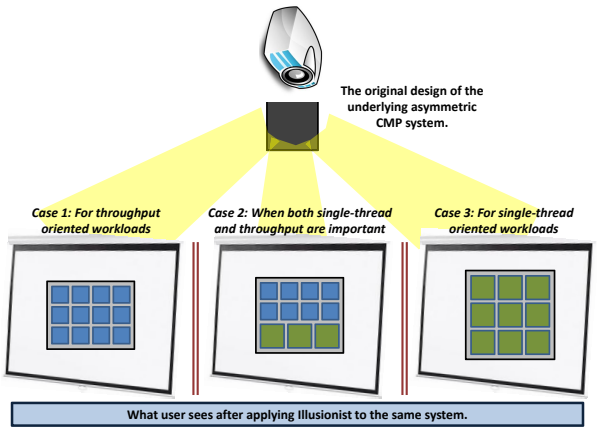



**Figure 1:** The high-level objective of Illusionist. In reality, we have a CMP with 12 cores. However, we want to give the system this ability to appear as a CMP system with any combination of aggressive and lightweight cores depending on the characteristics and requirements of workloads running on the system.

objective is to give the end-user the impression that this is a CMP system with any combination of lightweight and aggressive cores depending on the workload characteristics and requirements. Using a fixed hardware fabric, Illusionist provides the appearance of customization without the overheads of reconfigurable hardware or fused processors.

In Illusionist, the performance of a large number of lightweight cores is enhanced using help from a few existing aggressive cores. Illusionist dynamically reconfigures itself and provides a trade-off between single-thread performance and system throughput. To achieve this, we assign a group of lightweight cores to each aggressive core. The aggressive core is responsible for accelerating the threads (or processes in this context) that are running on all those lightweight cores. For this purpose, the aggressive core runs a modified version of all the programs that are running on all those lightweight cores and generates execution hints for the lightweight cores. The lightweight cores then receive these hints and use those to get a higher local cache hit rate and also higher branch prediction accuracy. However, the IPC difference between an aggressive and a lightweight core is not more than 2X. This simply means that an aggressive core cannot execute all the threads that are running on a large number of lightweight cores simultaneously (e.g., 10 threads). In order to make this happen, we take two main approaches to generate a second set of programs from the original set running on the lightweight cores. These programs are much shorter while having the same structure as the original programs.

To distill a program, we first perform code analysis in a dynamic compiler and remove all the instructions that are not necessary for generating the execution hints by an aggressive core. Next, we design a predictor which can predict during which program execution phases a lightweight core can benefit the most from the execution hints. By putting these two techniques together, on average, we were able to remove about 90% of the original instructions while taking a minimal hit on how much a lightweight core could potentially be accelerated with perfect hints.

The primary contributions of this paper are: **1)** Proposing an adaptive asymmetric multicore system that can dynamically achieve a higher single-thread performance for a large number of lightweight cores by appropriately sharing an aggressive core among them; **2)** A just-in-time compiler analysis to significantly distill a program without impacting its original structure in terms of hint generation; **3)** An application-specific predictor for identifying the most fruitful phases of program execution based on the acceleration that can be seen by a lightweight core; and **4)** An extensive comparison between our proposed scheme and a system with all lightweight cores and a system with all aggressive cores in terms of single-thread performance and system throughput.

## 2 Motivation

### 2.1 Acceleration Opportunities

One of the fundamental concepts in the design of Illusionist is the coupling of cores. As mentioned, we couple a lightweight core to an aggressive with the purpose of accelerating the execution of the lightweight core through the hints generated by the aggressive core. Both cores, in a basic case, run the exact same program. Here, we want to show by exploiting hints from an aggressive core, the lightweight core can typically achieve a significantly higher performance. In the rest of the paper, we will discuss the set of techniques that we used to make the coupling of the two cores more effective by providing intrinsically robust hints and effective hint disabling to ensure the lightweight core is not mislead by unprofitable hints.

For the purpose of evaluation and since we want to have a single ISA system, based on the availability of the data on the power, area, and other characteristics of microprocessors, we use an EV6 (DEC Alpha 21264 [14]) for an aggressive core. On the other hand, for a lightweight core, we select a simpler core like the EV4 (DEC Alpha 21064) or EV5 (DEC Alpha 21164) to save on the overheads of adding this extra core to the CMP system. We use Alpha processors due to the availability of models, simulators, microarchitectural design details, and data on power, area, and performance. In order to evaluate the efficacy of the hints, in Figure 2, we show the performance boost for the aforementioned DEC Alpha cores using perfect hints (PHs) – perfect branch prediction and no L1 cache miss. Here, we have also considered the EV4 (OoO), an OoO version of the 2-issue EV4, as a potential option for our lightweight core. As can be seen, by employing perfect hints, the EV4 (OoO) can outperform the 6-issue OoO EV6 in most cases. This implies that it might be possible to achieve a performance close to that of an aggressive core through core coupling. Nevertheless, this is not an easy goal to achieve when we want to couple a single aggressive core to a large number of lightweight cores as the aggressive core simply cannot keep up with all the other cores.

### 2.2 Main Design Challenges

We start from the basic coupled core design and through the rest of this paper gradually build a system that can out-

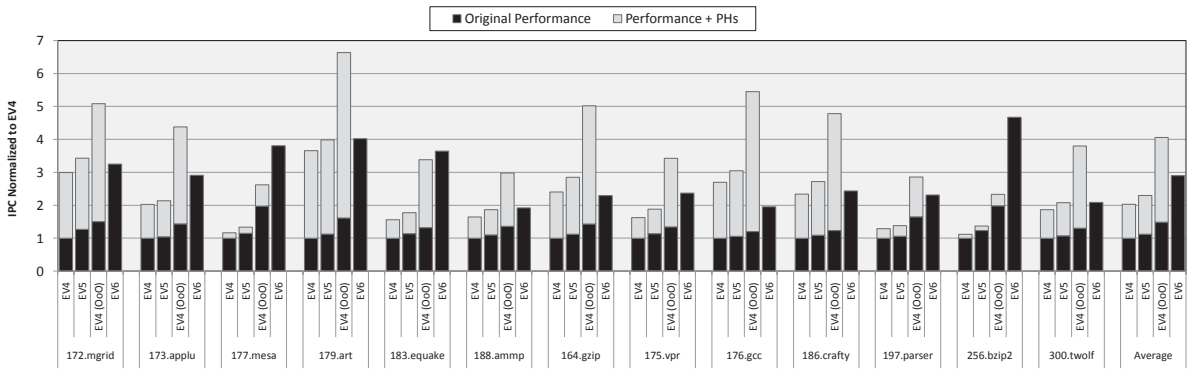



**Figure 2:** IPC of different DEC Alpha microprocessors, normalized to EV4's IPC. In most cases, by providing perfect hints for the simpler cores (EV4, EV5, and EV4 (OoO)), these cores can achieve a performance comparable to that achieved by a 6-issue OoO EV6.

perform the conventional CMP designs. In a simple coupled core design, each lightweight core is shared among multiple aggressive cores. This means one aggressive core, at any given point in time, runs the same program as the lightweight core. By replicating this design, we will have a CMP system in which each lightweight core is assigned to a single aggressive core. The objective of each aggressive core is to accelerate its corresponding lightweight core by running the same program and generating useful hints. Using the system configuration described in Table 1, on average, we can achieve 52.1% speedup for each lightweight core. However, for the same area budget, as we will see later in this work, the overall system throughput is 39.2% lower than a system that only consists of aggressive cores.

This means that we cannot expect to assign one aggressive core to each lightweight core. This design simply would not be cost effective as the performance of a coupled core is less than the original performance of an aggressive core. Therefore, we need to enhance the usefulness of the aggressive cores by effectively sharing them across a larger number of lightweight cores. However, since the IPC of an aggressive core is only 30% better than an accelerated lightweight core, time multiplexing the aggressive cores among lightweight cores is obviously ineffective. In the reminder of this paper, we describe how to increase the effectiveness of an aggressive core during hint generation process so that it can support more than a single lightweight core.

# 3 Illusionist

Here, we describe techniques that are used by Illusionist to distill programs running on the CMP system. In addition, we also demonstrate architectural and microarchitectural augmentations needed by Illusionist to achieve single-thread performance adaptability and higher throughput in more detail.

## 3.1 Program Distillation

Given a single aggressive core, in order to generate hints for a larger number of LightWeight Cores (LWCs), we distill the original program to a shorter version. This distilled program can run faster on an aggressive core. Therefore, the aggressive core can be shared among a larger number of LWCs through simple, multithreading-based time multiplexing. During this program distillation process, our objective is to perform an aggressive instruction removal while preserving the accuracy of the original hints.

### 3.1.1 Instruction Removal

In order to perform an aggressive instruction removal while preserving the effectiveness of the hints, we need to limit the instruction removal to the instructions that do not contribute to the generation of the hints. Since our hints are based on cache and branch misses, we only need to focus on preserving the memory operations and branch instructions. However, this cannot be done by removing all other instructions since there are dependencies between instructions. For example, to have the address of a load instruction, we need to keep an ALU operation which generates the address. Therefore, in our analysis, we start from branch and memory instructions and trace back the dependencies inside a window of instructions. These dependencies can be either register dependencies or memory dependencies. We perform analysis on both type of dependencies through back slices.

To perform this analysis, we define an analysis window which is a parameter in our system. This analysis window is a sliding window which covers a fixed number of instructions. At every step, this window slides one instruction which means one instruction will be added to this window from the bottom and one instruction will leave this window from the top. If the left out instruction does not, directly or indirectly, produce a value for any branch or memory operation inside the window, we remove the instruction from the program. Otherwise, that instruction will be intact in the distilled program.

Next, to remove more instructions, we start by removing highly biased branches. In our analysis, we remove any branch that has a bias higher than an empirical value of 90%. This means that we remove all the instructions inside the path corresponds to the 10% direction. Moreover, the back slice that represents the dependencies to those highly biased branches are also be removed from the distilled program. This affects the accuracy of the hints. However, since these branches are highly predictable, the branch predictor of the LWC should be able to provide an accurate prediction for

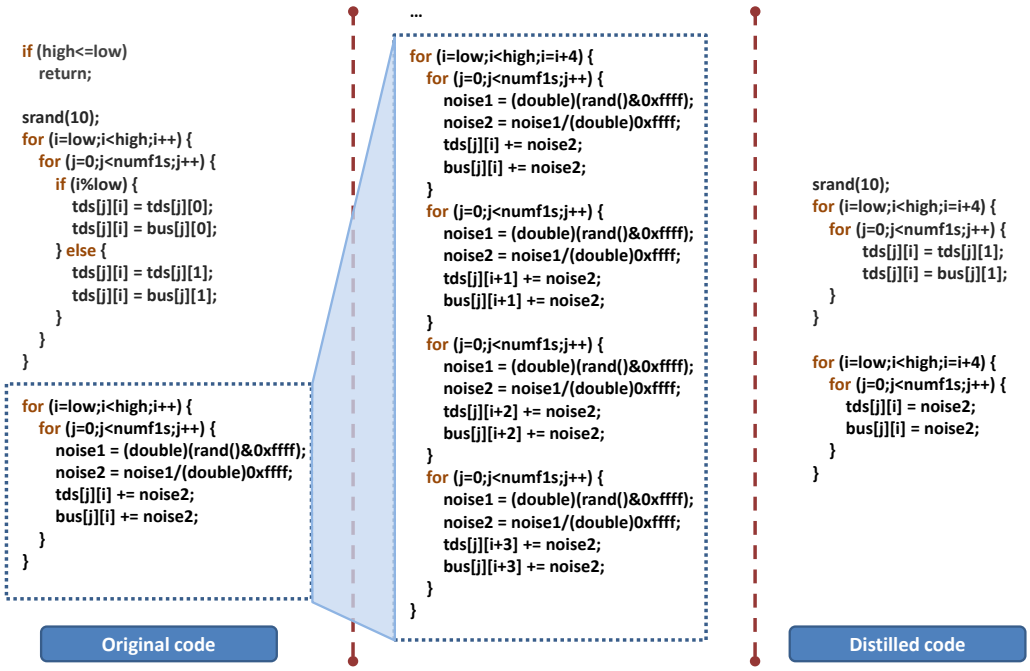



**Figure 3:** An example that illustrates our instruction removal process. At the left most column, we have a code segment from 179.art. To make the example easier to understand, in the middle column, we show the bottom part when the outer loop is unrolled 4 times (the number of doubles that fit into a single cache line). The right most column shows the same code after performing our instruction removal. As you can see, since the first if statement is executed very rarely, it has been removed from the distilled program. Next, since the else clause is highly biased, we remove its corresponding if clause. Moreover, we perform a loop unrolling on the outer loop and get rid of second, third, and fourth inner loops as they only have redundant cache line references. Finally, for the last nested loops, we perform the same unrolling and remove all the inner loops except the first one. Here, since the data by itself is not important for the hint generation, we got rid of the value assignments to *noise1* and *noise2* plus their additions to *tds* and *bus* elements.

these branches. This means the hints that are related to these branches do not have a notable impact on the lightweight core's acceleration.

Another approach that we take to get a more effective distilled program is to remove the unnecessary cache hints. There are many scenarios in which a cache hint tries to bring the same line to the cache. To avoid this, we perform an analysis on the addresses of loads and stores inside the analysis window and remove loads and stores with an address which hits the same cache line. In other words, if there is a load that accesses address $Address_1$ and there is a store, later in inside the same window, that accesses address $Address_2$, if both $Address_1$ and $Address_2$ are in the same cache line, we remove the store instruction. Using all the mentioned approaches, we remove most instructions that generate ineffective branch, I-cache, D-cache, and L2 cache hints. As we will see, this significantly reduces the size of the distilled program while preserving most of the benefits. Figure 3 shows an example of our instruction removal process. Program distillation could be performed offline. However, our objective is to operate on unmodified binaries and generate different hints based on the phase of the program. To perform this analysis on the fly, we propose using DynamoRIO in our system. A dynamic, just-in-time compiler that has the performance overhead of less than one percent [7, 1]. This based on the fact that we have a constant analysis window size and our program distillation consists of multiple linear passes in the dynamic compiler with the time complexity of O(n).

### 3.1.2 Phase-Based Program Selection

An orthogonal approach to increase the utilization of the aggressive cores is to consider program phase behavior. Programs have different phases when focusing on IPC, branch prediction accuracy, or cache miss rate. Therefore, the amount of benefit that a LWC can get from the hints varies based on what phase it is currently at. Figure 4 shows the performance of an accelerated LWC for a single benchmark. The performance is shown across 15 million instructions which covers multiple of these phases. As can be seen, the hint usability phases vary in length but they are mostly longer than half a million instructions.

If we can predict these phases without actually running the program on both lightweight and aggressive cores, we can only limit the dual core execution to the most useful phases. This means during the phases that the LWC does not benefit from the hints; we simply run threads from other LWCs on the aggressive core. Here, we design a simple but effective software-based predictor that can estimate the IPC of a LWC after receiving hints regarding to the program that is currently running. This predictor can sit either in the hypervisor-level or operating system and monitors the performance counters while the threads running. Given the cache miss rates, branch mispredictions, and IPC of a thread in the last epoch, our predictor uses a linear regression model for the estimation of IPC after acceleration. At a given point in time, an aggressive core runs the the thread that our predictor picks as the most suitable thread for the coupled core system. The predictor determines the thread

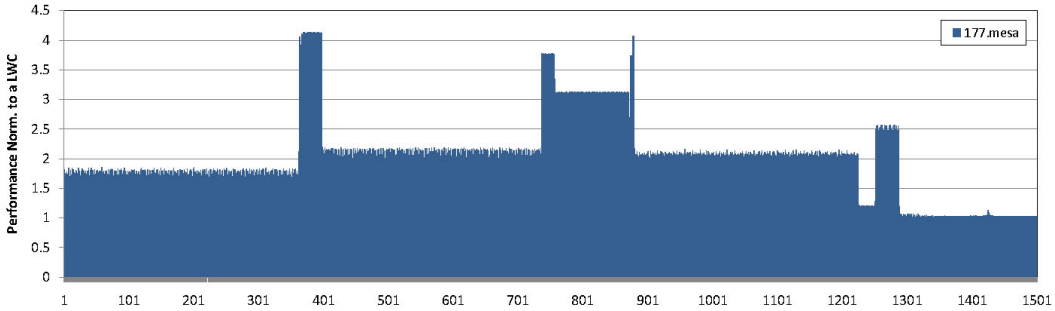



**Figure 4:** Performance of an accelerated lightweight core, for 177.messa, normalized to the original performance of a lightweight core. Each bar in this plot represents the average performance across 10K instructions and this performance is shown across 15 million instructions. As can be seen, there are phases or periods of time in which the hints from the aggressive core are more useful.

that is most likely to benefit every 100K instructions. Given that the acceleration phases are in the order of several million instructions, more frequent prediction is unnecessary. In addition, context switching every 100K instructions imposes a negligible performance overhead [24] that is accounted for in our evaluation.

As mentioned, our hints consist of L1 instruction cache hints, L1 data cache hints, L2 cache hints, and branch prediction hints. Therefore, for designing the predictor, we use these set of parameters: IPC of the LWC, L1 instruction cache miss rate, L1 data cache miss rate, core-specific L2 miss rate, and branch prediction miss rate. After that we perform a learning process on a data set gathered across a set of benchmarks (i.e., our training set) to train our regression model. In our training experiments, we noticed the absolute error in the prediction, of IPC after receiving hints, varies from 8% when using 100% of benchmarks for training to 15% when using 10% of benchmarks for training. For our evaluation, we use 50% of benchmarks, which are randomly selected from a set of SPEC-CPU-2K benchmark suite. We get around 10% absolute error in our IPC prediction by using the regression-based predictor. Figure 5 shows the performance of our predictor for 188.ammp across 40 million instructions. As can be seen in this figure, although there is error in the absolute value of the prediction, the predictor does a decent job on following the IPC trend after acceleration.

## 3.2 Illusionist Architecture

Illusionist employs a robust and flexible heterogeneous core coupling technique. Given a group of cores, we have an aggressive core, with the same ISA, that is shared among many lightweight cores. Here, we first describe the architectural details for a coupled pair of aggressive and lightweight cores. Later on this section, we describe the system for a CMP with larger number of lightweight cores. Given this base case, by executing the program on the aggressive core, Illusionist provides hints to accelerate the lightweight core. In other words, the aggressive core is used as an external run-ahead engine for the lightweight core. Figure 6 illustrates the high-level design of a coupled core execution technique. In our design, most communication is unidirectional from the aggressive core to the lightweight core with the exception of the resynchronization and hint disabling signals. Consequently, a single queue is used for sending the hints and cache fingerprints to the lightweight core. The hint gathering unit attaches a tag to each queue entry to indicate its type. When this queue gets full and the aggressive core wants to insert a new entry, it stalls. To preserve correct memory state, we do not allow the dirty lines of the aggressive core's data cache to be written back to the shared L2 cache. Furthermore, exception handling is also disabled within the aggressive core since the lightweight core maintains the precise state.

In this design, we do not rely on overclocking the aggressive core. Furthermore, this is a hardware-based approach that is transparent to the workload and operating system. It also does not require register file checkpointing for performing exact state matching between two cores. Instead, we employ a fuzzy hint disabling approach based on the continuous monitoring of hint effectiveness, initiating resynchronizations when appropriate. In order to make the hints more robust against microarchitectural differences between the two cores and also variations in the number/order of executed instructions, we leverage the number of committed instructions for hint synchronization and attach this number to every queue entry as an *age tag*. Moreover, we introduce the concept of a *release window* to make the hints more robust in the presence of the aforementioned variations. The release window helps the lightweight core determine the right time to utilize a hint. For instance, assuming the data cache release window is 20, and 1030 instructions have already been

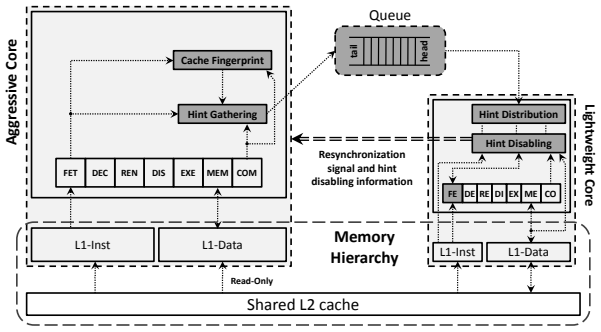



**Figure 6:** The high-level architecture of Illusionist for connecting a aggressive and a lightweight core together. Here, we highlighted the modules that are added or modified by Illusionist in the original system.

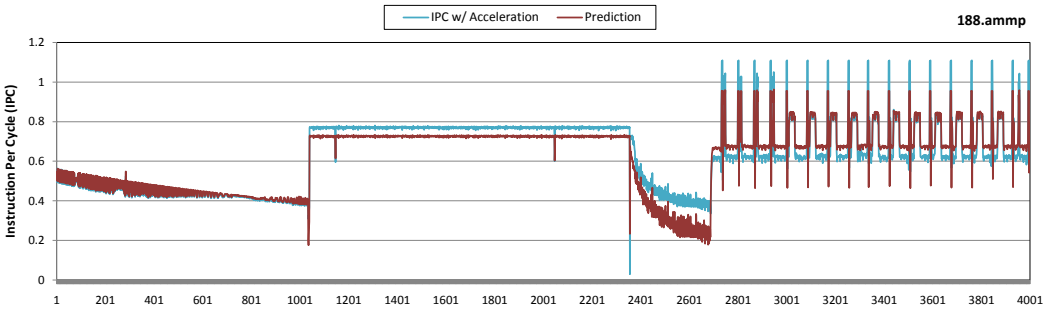



**Figure 5:** Here, the blue line shows the performance of a lightweight core plus acceleration during the execution of 40 million instructions in 188.ammp benchmark. Each bar in this plot represents the average performance across 10K instructions. The red line shows the predicted performance of the lightweight core after it receives hints from an aggressive core during the execution of the same benchmark.

committed in the lightweight core, data cache hints with age tags $\leq 1050$ can be pulled off the queue and applied.

### 3.2.1 Hint Gathering and Distribution

Our branch prediction and cache hints (except L2 prefetches) need to be sent through the queue to the lightweight core. The hint gathering unit in the aggressive core is responsible for gathering hints and cache fingerprints, attaching the age and type tags, and finally inserting them into the queue. The PC of committed instructions and addresses of committed loads and stores are considered as cache hints. Although a prefetcher might be easier to implement instead of applying cache hints in a lightweight core, it is known that the memory access patterns in most commercial and scientific workloads are often highly irregular and not amenable to simple predictive schemes [4]. In contrast, our scheme handles data-dependent addresses and it can focus on addresses that are likely to result in cache misses.

For branch prediction hints, the hint gathering unit sends a hint through the queue every time the branch predictor (BP) of the aggressive core gets updates. On the lightweight core side, the hint distribution unit receives these packets from the queue and compares their age tag with the local number of committed instructions plus the corresponding release window sizes. It treats the incoming cache hints as prefetching information to warm-up its local caches.

The default BP of the lightweight core is a bimodal predictor. We first add an extra bimodal predictor (Illusionist BP) to keep track of incoming branch prediction hints. Furthermore, we employ a hierarchical tournament predictor to decide, for a given PC, whether the original or Illusionist BP should take over. As mentioned earlier, we leverage release windows to apply the hints just before they are needed. However, due to the variations in the number of executed instructions on the aggressive core, even the release window cannot guarantee perfect timing of the hints. In such a scenario, for a subset of instructions, the tournament predictor can give priority to the original BP of the lightweight core to avoid any performance penalty. Figure 7 shows a simple example in which the Illusionist BP can only achieve 33% branch prediction accuracy. This is mainly due to the existence of a tight inner loop with a low trip count for which switching to the original BP can enhance the branch prediction accuracy.

In order to reduce the queue size, communication traffic needs to be limited to only the most beneficial hints. Consequently, in the hint gathering unit, we use two content addressable memories (CAMs) with several entries to discard I-cache and D-cache hints that were recently sent. Eliminating redundant hints also minimizes resource contention on the lightweight core. To save on transmission bits, we only send the block related bits of the address for cache hints and ignore hints on speculative paths. In addition, for branch prediction hints, we only send lower-order bits of the PC that are used for updating the branch history table of the Illusionist BP.

Instruction cache and branch prediction hints require correspondence between the program counter (PC) values in the distilled and original programs. This is needed to determine the right time to apply a particular hint. One option is to create a translation table to convert PC values. Instead, we choose to maintain identical starting addresses for all functions and basic blocks in the distilled program. This is accomplished by first removing the ineffectual instructions in each basic block, as described in Section 3.1.1. Then, inter-instruction white space is eliminated and instructions are justified to the start of the basic block. The collective white space resulting from instruction removal is then inserted as a block at the end of each basic block, thereby ensuring the starting address for the next basic block is not altered. To ensure proper control flow, an unconditional branch is inserted after the last instruction in the basic block to the next block to skip over the white space if at least one instruction was deleted. This process essentially maintains a skeleton control flow graph whose PC values are preserved in the distilled program. Note that PC preservation is not perfect and the PC values for instructions that are not eliminated can change. However, the maximum change for a single PC value is the size of the largest basic block, which for control-intensive programs is small. The LWC can simply mask off a few of the lower bits of the addresses contained in the hints to hide these discrepancies.

### 3.2.2 Why Disable Hints?

Hints can be disabled when they are no longer beneficial for the lightweight core. This might happen because the program execution on the aggressive core diverges from the correct execution path, performance of the lightweight core is already near its ideal case, or the aggressive core is not be

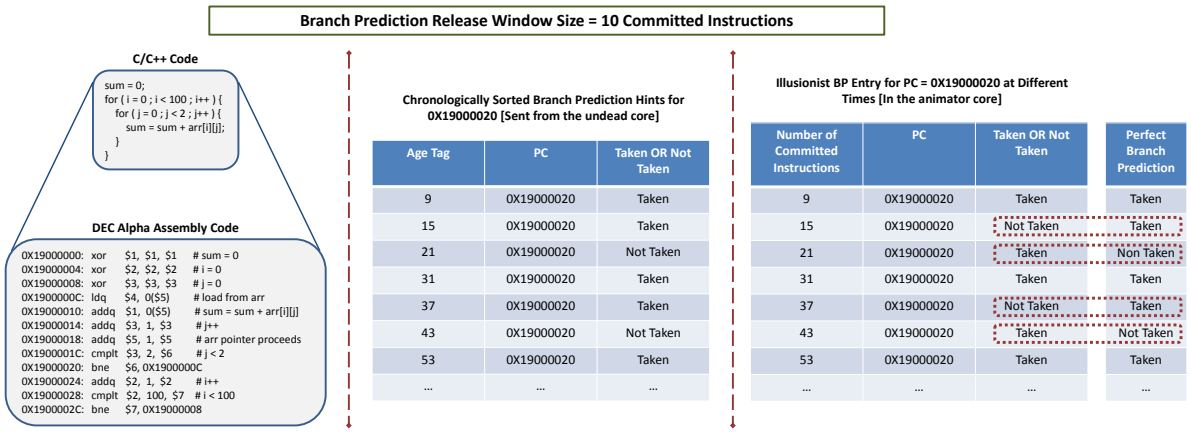



**Figure 7:** A code example in which hints are received by the lightweight core at improper times, resulting in low branch prediction accuracy. Therefore, switching to the original BP of the lightweight core is beneficial. The code simply calculates the summation of a 2D-array elements. It should be noted that the branch prediction release window size is normally set so that the branch prediction accuracy for the entire execution gets maximized.

able to get ahead of the lightweight core. In all these scenarios, hint disabling helps in four ways: (1) It avoids occupying resources of the lightweight core with ineffective hints. (2) The queue fills up less often, which means less stalls for the aggressive core. (3) Disabling hint gathering and distribution saves power. (4) It serves as an indicator of when the aggressive core has strayed far from the correct execution path and resynchronization is required.

### 3.2.3 Hint Disabling Mechanisms

The hint disabling unit is responsible for realizing when each type of hint should get disabled. In order to disable cache hints, the cache fingerprint unit generates high-level cache access information based on the committed instructions in the last time interval (e.g., last 1K committed instructions). These fingerprints are sent through the queue and compared with the lightweight core's cache access pattern. Based on a pre-specified threshold value for the similarity between access patterns, the lightweight core decides whether the cache hint should be disabled. In addition, when a hint gets disabled, the hint remains disabled throughout a significant time period, the back-off period. After this time passes, the cache hints will be re-enabled again.

Apart from prioritizing the original BP of the lightweight core for a subset of PCs, the Illusionist BP can also be employed for global disabling of the branch prediction hints. For disabling branch prediction hints, we use a score-based scheme with a single counter. For every branch that the original and Illusionist BPs agree upon no action should be taken. Nonetheless, for the branches that the Illusionist BP correctly predicts and the original BP does not, the score counter is incremented by one. Similarly, for the ones that the Illusionist BP mispredicts but the original BP correctly predicts, the score counter is decremented. Finally, at the end of each disabling time interval, if the counter is below a certain threshold, the branch prediction hints will be disabled for a back-off period.

### 3.2.4 Resynchronization

Since the aggressive core can stray from the correct execution path and no longer provide useful hints, a mechanism is required to restore it back to a valid state. To accomplish this, we occasionally resynchronize the two cores, during which the lightweight core's PC and architectural register values are copied to the aggressive core. According to [24], for a modern processor, this process takes on the order of 100 cycles. Moreover, all instructions in the aggressive core's pipeline are squashed, the rename table is reset, and the D-cache content is also invalidated. We take advantage of the hint disabling information to identify when resynchronization is needed. For instance, a potential resynchronization policy is to resynchronize when at least two of the hints get disabled.

### 3.2.5 Illusionist Applied to a Large CMP

Here, we describe how Illusionist can be applied to CMP systems with more cores. Figure 8 illustrates the Illusionist design for a 44-core system with 4 clusters. There are two reasons why we do not prefer to couple lightweight cores with each other in Illusionist. First, there is more than 2X IPC difference between a baseline lightweight core and an aggressive core. More importantly, since a lightweight core has limited resources (i.e., L1-data and instruction caches, BHT, and no RAS), there would be a much higher contention for the resources that need to be shared across all the distilled programs that are running on this lightweight core. Here, each cluster contains 11 cores consisting of 10 lightweight cores and a single aggressive core, shown in the call-out. Here, the number of lightweight cores that are assigned to a aggressive core is decided based on the execution power of the aggressive core given a randomly chosen sets of all distilled programs that it needs to run. In order to maintain scalability of the Illusionist design, we employ the aforementioned 11-core cluster design as the basic building block for larger CMP systems. The aggressive core generates the hints for all the processes or threads that are running on the lightweight cores in the same cluster and sends these hints through hint gathering unit as described before. The hint

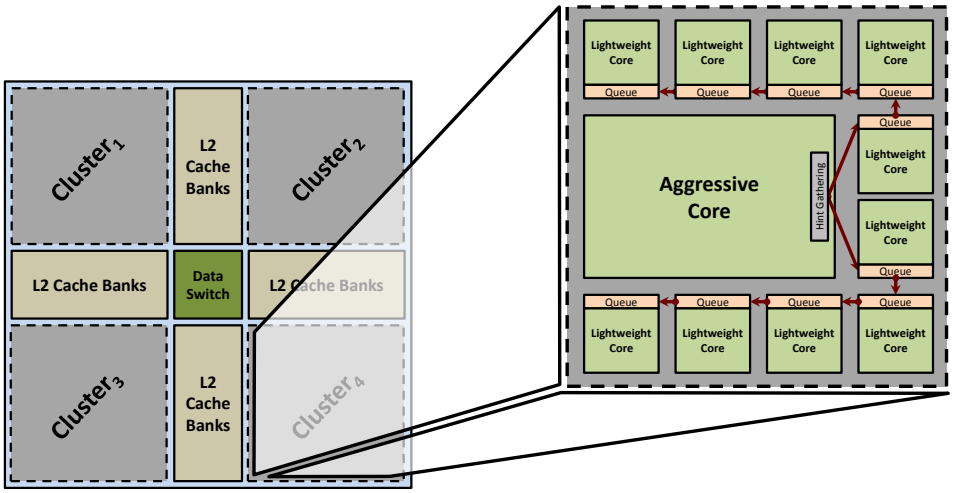



**Figure 8:** Illusionist system overview.

gathering unit is connected to two ring interconnection networks in a tree manner. First the hint gathering unit decides which ring the hint needs to be sent to and it will push the hint to the corresponding ring. Each lightweight core has a queue dedicated to the transfer of hints inside the ring and the snoop the queue to see whether they should pull a particular hint and apply it or they should forward the hint to the next queue in the same ring. Therefore, the last lightweight core in the ring will receive a hint after going through 4 other hubs. However, our analysis shows that the couple core execution technique is very robust against these communication latencies since the distilled code is executed on an aggressive core far in advance.

For a heterogeneous CMP system with different type of cores, the cores can be statically grouped such that the weaker cores in each group can benefit the most from the execution of their threads on a relatively stronger core. In such a scenario, we would like to partition the original set of cores into groups such that in each group, we have a few aggressive cores and a large number of lightweight cores. This design strategy is a suitable fit for many heterogeneous CMP systems that are designed with many simple cores such as the IBM Cell processor.

# 4 Evaluation

In this section, we go over the experimental setup and also describe experiments performed to evaluate the merit of Illusionist in increasing system's dynamic adaptability given different performance requirement scenarios.

## 4.1 Experimental Methodology

In order to evaluate the performance impact of Illusionist's heterogeneous execution, we heavily modified SimAlpha [5], a validated cycle accurate microarchitectural simulator. We have two flavors of the simulator, implementing the lightweight and also aggressive cores. We accurately simulate the generation and propagation of the correct and incorrect hints in the system. Inter-process communication has been used to model the information flow between different cores (e.g., L2 warm-up, hints, and cache fingerprints). Both instruction removal process and phase-based program selection are implemented in the simulator to allow dynamic analysis of the benchmarks running on our system. A 6-issue OoO EV6 and a 2-issue OoO EV4 are chosen as our baseline aggressive and lightweight cores, respectively. The configuration of these two baseline cores and the memory system is summarized in Table 1. We run the SPEC-CPU-2K benchmark suite cross-compiled for DEC Alpha and fast-forwarded to an early SimPoint [26]. The We used all programs in SPEC-CPU-2K that get cross-compiled with no issues and work out of the box with our simulation infrastructure. We assume both types of cores run at the same frequency as the amount of work per pipeline stage remains relatively consistent across Alpha microprocessor generations [17], for a given supply voltage level and a technology node. Nevertheless, our system can clearly achieve even a higher performance if the lightweight cores were allowed to operate at a higher frequency.

Dynamic power consumption for both types of cores is evaluated using an updated version of Wattch [3] and leakage power is evaluated with HotLeakage [31]. Area for our EV6-like core – excluding the I/O pads, interconnection wires, the bus-interface unit, L2 cache, and control logic – is derived from [17]. In order to derive the area for a lightweight core, we start from the publicly available area break-down for the EV6 and resize every structure based on the size and number of ports. Furthermore, CACTI [21] is used to evaluate the delay, area, and power of the on-chip caches. Overheads for the SRAM memory structures that we have added to the design, such as the Illusionist branch prediction table, are evaluated with the SRAM generator module provided by the 90nm Artisan Memory Compiler. Moreover, the Synopsys standard industrial tool-chain, with a TSMC 90nm technology library, is used to evaluate the overheads of the remaining miscellaneous logic (e.g., MUXes, shift registers, and comparators). Finally, the area for interconnection wires between the coupled cores is estimated using the same method-

**Table 1:** Configuration of the underlying lightweight and aggressive cores.

| Parameter | A lightweight core | An aggressive core |
|---|---|---|
| Fetch/issue/commit width | 2 per cycle | 6 per cycle |
| Reorder buffer | 32 entries | 128 entries |
| Load/store queue entries | 8/8 | 32/32 |
| Issue queue | 16 entries | 64 entries |
| Instruction fetch queue | 8 entries | 32 entries |
| Branch predictor | tournament (bimodal + Illusionist BP) | tournament (bimodal + 2-level) |
| Branch target buffer size | 256 entries, direct-map | 1024 entries, 2-way |
| Branch history table | 1024 entries | 4096 entries |
| Return address stack | - | 32 entries |
| L1 data cache | 8KB direct-map, 3 cycles access latency, 2 ports | 64KB, 4-way, 5 cycles access latency, 4 ports |
| L1 instr. cache | 4KB direct-map, 2 cycles access latency, 2 ports | 64KB, 4-way, 5 cycles access latency, 1 port |
| L2 cache | 1MB per core, unified and shared, 8-way, 16 cycles access latency | |
| Main memory | 250 cycles access latency | |

ology as in [18], with intermediate wiring pitch taken from the ITRS road map [13].

## 4.2 Experimental Results

In this section, we evaluate different aspects of our proposed design such as program distillation efficacy, single-thread performance improvement, area, power and energy overheads, and finally we take a look at our system throughput enhancement.

First, we start by looking at the amount of reduction in the code size during our instruction removal process. Figure 9 demonstrates the effectiveness of the approach discussed in Section 3.1.1. As can be seen, on average, for a window size of 10K instructions, around 75% of instructions can be removed. It should be noted that as the size of the window grows, in most cases, more instructions can be removed from the program. The reason behind this is that as the analysis window size increases, there are more memory instructions that alias and can be removed. However, in certain scenarios such as 172.mgrid and 177.mesa, as we increase the window size, the impact of register dependencies dominate memory alias. This causes a lower reduction in the size of the distilled program. Nevertheless, for most programs, the memory alias clearly dominates the register dependencies. Our analysis can remove more than 90% of 172.mgrid dynamic instructions. The main reason behind this is that 172.mgrid has relatively large hot loops with very limited and predictable memory accesses.

Next, we evaluate the accuracy of the hints compared to the case that no instruction is removed from the original program. Figure 10 demonstrates this result. As mentioned, larger window size results into more aggressive instruction removal; thus, as one would expect, this comes with a higher loss of accuracy. This is because of the fact that the more aggressive instruction removal gets, the more accuracy loss should be observed. For the rest of our evaluation, we use analysis window size of 10K instructions which offers a good trade-off between accuracy and percentage of instructions removed. Given this analysis window size, on average, 79% of all the hints (in the perfect program execution case) will be present in the subset of generated hints by the distilled program.

Another point that should be noted is the distribution of instructions after instruction removal. Figure 11 shows the distribution of instructions based on their type before and after instruction removal. As can be seen, in most scenarios, these distributions are almost the same. This means that there would not be a flood on the cache and branch prediction ports. In other words, the resource usage characteristics of these distilled programs are very close to the real benchmarks. For 177.messa, most of the remaining instructions after instruction removal are memory instructions. However, this does not put a notable overhead on the system as more than 80% of instructions are removed from this benchmark. Another interesting point is that most of float and miscellaneous instructions are removed from the floating point

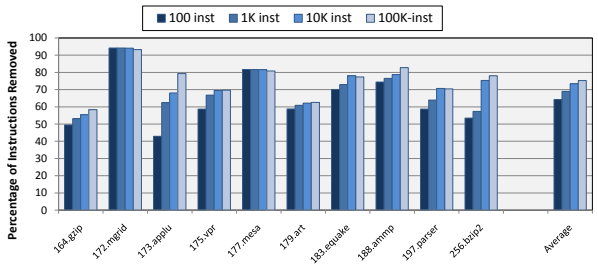



**Figure 9:** The percentage of instructions removed from the original program by our scheme. Here, we varied the analysis window from 100 instructions to 100K instructions and looked at the size of the distilled program. As can be seen, on average, 76% of instructions are removed by our scheme when performing the analysis on a sliding window of size 100K instructions.

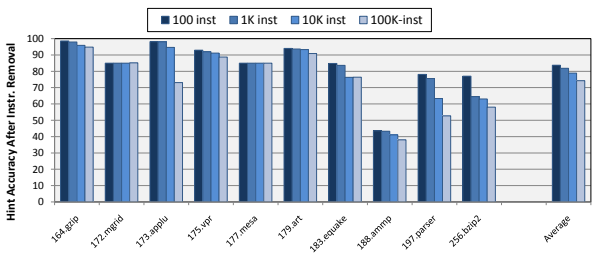



**Figure 10:** The accuracy of generated hints by the distilled program is shown here. We varied the analysis window from 100 instructions to 100K instructions and looked at the accuracy of the hints. As expected, the accuracy of the hints drop as the window size grows.

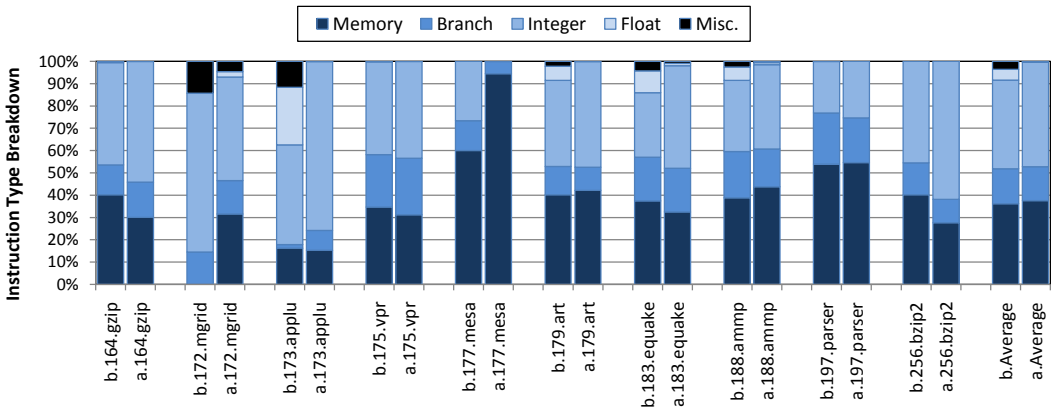



**Figure 11:** Breakdown of instructions by their type. Here, instructions are categorized as memory, branch, integer, float, or miscellaneous operations. This breakdown is represented before **(b)** and after **(a)** program distillation to show how our instruction removal process affects the distribution of instructions in the distilled program.

benchmarks such as 173.applu as they do significantly not contribute to the hint generation process.

The important question here is how much performance we can get out of a LWC after we performed our instruction removal. Figure 12 shows this result. There are three sets of bars here. The first set represents the performance of a LWC for a given benchmark. The second set is the performance of a LWC after receiving hints from an aggressive core running the distilled program. Finally, the third set shows the performance of an aggressive core. Given a particular benchmark, all these results are normalized to the performance of a LWC. On average, the LWC can get 43% speedup after receiving hints from the execution of the distilled program on an aggressive core.

Finally, Figure 13 presents an area-neutral comparison between our proposed scheme and two extremes of conventional CMP design. This study is performed for different levels of thread availability in the system. As shown in Figure 13(a), at one end of the spectrum, we have a CMP system with all aggressive cores (with 21 cores) and at the other end, we have a CMP system with all LWCs (with 81 cores). We show three other alternatives here. First one is a system in which every LWC is assigned to a single aggressive core. As can be seen, for a scenario with high number of available threads, such a system achieves a poor system throughput

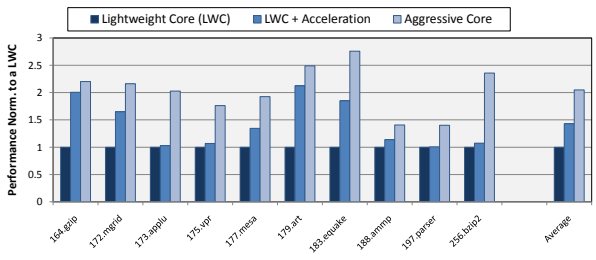
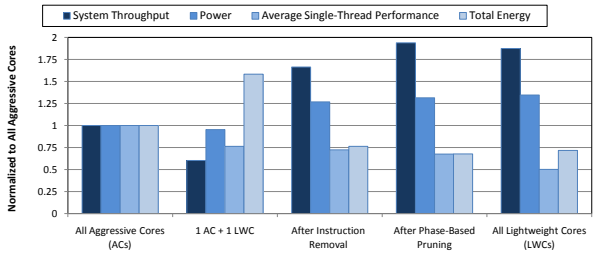



**Figure 12:** Performance of a lightweight core, lightweight core plus acceleration, and also an aggressive core across benchmarks are shown here. All the performances are normalized to a lightweight core's performance. The performance after acceleration represents the case that the original program is running on the lightweight core and the distilled program is running on an aggressive core.

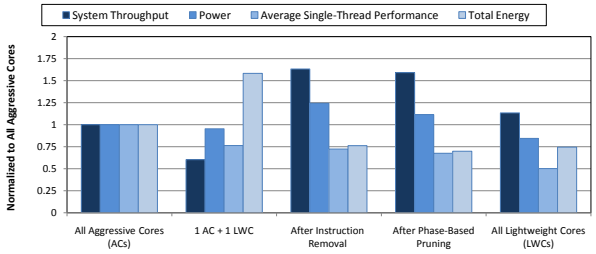
as aggressive cores do not contribute much to the system throughput. The second alternative is our scheme after instruction removal process. At last, we have our proposed scheme with instruction removal and phase-based program pruning. Compared to a CMP system with all LWCs, our system achieves a marginally better throughput and power consumption while it achieves 34.6% better single-thread performance. On the other hand, compared to a CMP system with all aggressive cores, our scheme with instruction removal and phase-based program pruning can achieve almost 2 times better throughput. For a scenario with medium and low number of available threads, as shown in Figure 13(b) and Figure 13(c), our system after phase-based pruning drastically outperforms a system with all LWCs both in terms of single-thread performance and system throughput. However, in scenarios with very low number of available threads, a system with all aggressive cores clearly performs better than all other alternatives. Nevertheless, even in such a scenario, Illusionist's throughput is higher than 80% of that of a system with all aggressive cores.

## 5 Related Work

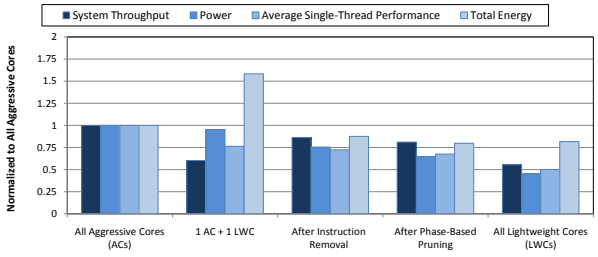
To the best of our knowledge, Illusionist is the first work that uses a single aggressive core to accelerate the execution of many lightweight cores. Prior work has shown two cores can be coupled together to achieve higher single-thread performance. Since the overall performance of a coupled core system is bounded by the slower core, these two cores were traditionally identical to sustain an acceptable level of single-thread performance. To accelerate program execution, one of these coupled cores must progress through the program stream faster than the other. Paceline [8] uses different frequencies for the core that runs ahead and the core that receives execution hints. By cutting the frequency safety margin of the core that running ahead, it tries to achieve a better single-thread performance for that core. Flea-Flicker two pass pipelining [2] allow the leader core to return an invalid value on long-latency operations and proceed. Illusionist applies a different set of microarchitectural techniques to gather hints, determine hints' timings, perform fine-grained hint disabling, and fast resynchronization. Moreover, the ob-




**(a)** number of available threads = number of LWCs




**(b)** number of available threads = 60% of the number of LWCs




**(c)** number of available threads = 30% of the number of LWCs

**Figure 13:** Area-neutral comparison between the system throughput, power, average single-thread performance, and total energy of our proposed scheme and baseline CMP systems. Here, we compared a CMP system with all aggressive cores (consisting of 21 cores) with several alternatives. To show the advantage of our scheme, this comparison is performed for three different system utilization levels (with 81, 48, and 24 available threads, respectively).

jective of our work is much broader than accelerating a single core using another identical core.

Program distillation is a technique that has been performed in both hardware and software. Slipstream processors [25] tries to accelerate sequential programs by connecting two processors through a queue. In slipstream processors, the A-stream runs a shorter version of the program that R-stream runs. The A-stream is inherently speculative and is dynamically formed during the execution of the program. There are predictors to help identifying ineffective instructions and removing them during fetch, execution, and retirement stages. In contrast, Master/Slave speculative parallelization [32] (MSSP) generates two different versions of the same program during the compilation phase. However, the objective of this work is to increase the single-thread performance of a single aggressive core by using speculations, which will be verified later in time, using other lightweight cores on the same CMP. Moreover, the program distillation techniques used in this work are not as aggressive as the ones introduced by Illusionist. On average, less than 25% of dynamic instructions can been removed by MSSP's program distillation techniques. The main reason that Illusionist can remove a significant portion of the dynamic instructions lays on the fact that Illusionist can operate robustly even in the presence of high execution inaccuracies. In Accelerated Critical Sections [29], critical sections are executed by an aggressive core to reduce the wait time of other processors. Therefore, it reduces the average execution time of parallel benchmarks with coarse critical sections.

Sherwood et. al. proposed a phase tracking and prediction scheme which allows capturing the phase-based behavior of a program on a large time scale [27, 26]. They keep track of execution frequencies of different basic blocks during a particular execution interval. Next, a basic block vector is build from these frequencies to represent a program phase. A phase change happens when the Manhattan distance between two adjacent basic block vectors exceeds a pre-specified threshold. Another scheme has been proposed by Huang et. al. [11] which uses a hardware-based call stack to identify program phases. This stack keeps track of the time spent in each subroutine. If this time exceeds a certain threshold, it will be identified as a major phase. In contrast to prior work, our objective is to predict how much performance enhancement can a lightweight core achieve after getting coupled to an aggressive core. In addition to this high-level objective difference, we also need to perform the prediction for relatively short intervals which comes with different challenges.

An alternative approach to achieve a higher single-thread performance on-demand is to use reconfigurable CMP architectures. Core fusion [12] is an example of these schemes which merges the resources of multiple adjoining single-issue pipelines to form large superscalar processors. Core fusion has a different design philosophy. Unlike Illusionist, it merges multiple cores to achieve single-thread performance for a single virtual core. For this purpose, significant modifications to the design of underlying cores and the multi-core chip is required. Moreover, to allow fusion among cores, it introduces several centralized resources for fetch management, register renaming and instruction steering. Other examples of this class of solutions are Core Federation [30], Composable Lightweight Processors [15], and CoreGenesis [9] which introduce an architecture that can combine neighboring scalar cores with or without the aid of shared structures to form super-scalar processors.

## 6 Conclusion

Dealing with power dissipation has become a major challenge in the way of getting higher performance in modern microprocessors. Consequently, there has been a major shift toward CMP systems with relatively simple cores. These lightweight cores do not have the ability to provide a descent single-thread performance for most applications. Usage of asymmetric CMPs is a positive step toward getting a better single-thread performance for a small fraction of threads running on the system. To achieve a higher and more uniform single-thread performance for all the threads that are running on the system, in this work, we presented Illusionist, which dynamically reconfigure the underlying CMP based on the desired trade-off between single-thread performance

and system throughput. As we discussed, this is achieved by accelerating the lightweight cores on the chip using a few existing aggressive cores. The main challenge here was that the number of threads that can be simultaneously run on an aggressive core is limited to at most 2 given the IPC difference between an aggressive and a lightweight core. As a result, a set of dynamic instruction stream and phase-based program analysis were presented to allow a more intelligent coupling of lightweight cores to a particular aggressive core. In other words, our target was to shrink the original program, that runs on a lightweight core, as much as possible while preserving its original structure so that the generated hints can still significantly help the coupled lightweight cores. Finally, we observed that Illusionist provides an interesting design that achieves 35% better single thread performance for all the threads running on the system compared to a CMP with all lightweight cores while achieving almost 2X better throughput compared to a CMP with all aggressive cores.

## References


[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proc. of the '00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[2] R. D. Barnes, E. N. Nystrom, J. W. Sias, S. J. Patel, N. Navarro, and W. W. Hwu. Beating in-order stalls with "flea-flicker" two-pass pipelining. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, page 387, 2003.

[3] D. Brooks, V. Tiwari, and M. Martonosi. A framework for architectural-level power analysis and optimizations. In *Proc. of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[4] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proc. of the '02 Conference on Programming Language Design and Implementation*, pages 199–209, 2002.

[5] R. Desikan, D. Burger, and S. Keckler. Measuring experimental error in microprocessor simulation. In *Proc. of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, 2001.

[6] J. Friedrich et al. Desing of the power6 microprocessor, Feb. 2007. In *Proc. of ISSCC*.

[7] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

[8] B. Greskamp and J. Torrellas. Paceline: Improving single-thread performance in nanoscale cmps through core overclocking. In *Proc. of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 213–224, 2007.

[9] S. Gupta, A. Ansari, S. Feng, and S. Mahlke. Stageweb: Interweaving pipeline stages into a wearout and variation tolerant cmp fabric. In *Proc. of the 2010 International Conference on Dependable Systems and Networks*, June 2010.

[10] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, July 2008.

[11] M. C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 157–168, 2003.

[12] E. Ipek, M. Kirman, N. Kirman, and J. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, pages 186–197, 2007.

[13] ITRS. International technology roadmap for semiconductors 2008, 2008. http://www.itrs.net/.

[14] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[15] C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 381–393, Dec. 2007.

[16] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, Feb. 2005.

[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, N. Jouppi, and D. Tullsen. Conjoined-core chip multiprocessing. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 195–206, 2004.

[19] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *IEEE Computer*, 38(11):32–38, Nov. 2005.

[20] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *Computer Architecture Letters*, 5(1):4–, Jan. 2006.

[21] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *IEEE Micro*, pages 3–14, 2007.

[22] U. Nawathe et al. An 8-core, 64-thread, 64-bit, power efficient SPARC SoC (Niagara2), Feb. 2007. In *Proc. of ISSCC*.

[23] K. Olukotun, B. A. Nayfeh, L. Hammond, K. G. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, 1996.

[24] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, June 2009.

[25] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A study of slipstream processors. In *Proc. of the 33rd Annual International Symposium on Microarchitecture*, pages 269–280, 2000.

[26] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, New York, NY, USA, 2002. ACM.

[27] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 336–347, 2003.

[28] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The impact of technology scaling on lifetime reliability. In *Proc. of the 2004 International Conference on Dependable Systems and Networks*, pages 177–186, June 2004.

[29] 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.

[30] D. Tarjan, M. Boyer, and K. Skadron. Federation: Repurposing scalar cores for out-of-order instruction issue. In *Proc. of the 45th Design Automation Conference*, June 2008.

[31] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects. Technical report, Univ. of Virginia Dept. of Computer Science, Jan. 2003.

[32] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 85–96, Nov. 2002.