Appears in Intl. Symp. on High Performance Computer Architecture, San Francisco, CA, Feb. 2015

XChange: A Market-based Approach to Scalable Dynamic Multi-resource Allocation in Multicore Architectures

Xiaodong Wang and José F. Martínez

Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA

http://m3.csl.cornell.edu/

ABSTRACT

Efficiently allocating shared on-chip resources across cores is critical to optimize execution in chip multiprocessors (CMPs). Techniques proposed in the literature often rely on global, centralized mechanisms that seek to maximize system throughput. Global optimization may hurt scalability: as more cores are integrated on a die, the search space grows exponentially, making it harder to achieve optimal or even acceptable operating points at run-time without incurring significant overheads.

In this paper, we propose XChange, a novel CMP resource allocation mechanism that delivers scalable high throughput and fairness. Through XChange, the CMP functions as a market, where each shared resource is assigned a price which changes over time, and each core seeks to maximize its own utility, by bidding for these shared resources. Because each core works largely independently, the resource allocation becomes a scalable, mostly distributed decision-making process. In addition, by distributing the resources proportionally to the bids, the system avoids unfairness, treating each core in an unbiased manner.

Our evaluation shows that, using detailed simulations of a 64-core CMP configuration running a variety of multiprogrammed workloads, the proposed XChange mechanism improves system throughput (weighted speedup) by about 21% on average, and fairness (harmonic speedup) by about 24% on average, compared with equal-share on-chip cache and power distribution. On both metrics, that is at least about twice as much improvement over equal-share as a state-of-the-art centralized allocation scheme. Furthermore, our results show that XChange is significantly more scalable than the state-of-the-art centralized allocation scheme we compare against.

1. INTRODUCTION

Designing chip multiprocessors (CMPs) that scale to more than a handful of cores is an important goal for the upcoming technology generations. A challenge to scalability is the fact that these cores will inevitably share hardware resources, whether it be on-chip storage, memory bandwidth, the chip's power budget, etc. Studies have shown that allowing cores to freely contend for shared resources can harm system performance [3, 9, 10]. Therefore, allocating resources efficiently among cores is key to achieving good behavior.

One challenge in resource allocation is that it is generally not a problem separable by resource, as resource interactions exist [3]. For example, increasing an application's allocated cache space may reduce its memory bandwidth demand, due to the lower cache miss rate. Similarly, increasing an application's power budget could allow it to run at a higher frequency, potentially demanding higher memory bandwidth. As more and more cores are integrated on a single die, the size of this multi-resource allocation space explodes, making it harder to devise mechanisms to lock on a good allocation without incurring significant overheads. Although prior knowledge of the applications from offline profiling may curb some of the run-time overhead, this information is generally not available.

An additional important consideration is the balance between throughput and fairness. Eyerman and Eeckhout [14] argue that a good resource allocation scheme should be able to maintain a balance between single-program performance and overall system throughput. However, existing global optimization solutions deal primarily with system throughput [9, 10], potentially resulting in systems with poor fairness. On the other hand, a recently proposed proportional allocation technique by Zahedi and Lee focuses on guaranteeing strict game-theoretic fairness of the co-running applications [48], but its formulation may come in practice at a cost in throughput, as we show in our results.

Contributions

In this paper, we propose XChange, a novel CMP multi-resource allocation mechanism that is able to deliver scalable high throughput and fairness. We formulate the problem as a purely dynamic, largely distributed market, where the "prices" of resources are adjusted based on supply and demand. Cores dynamically learn their own resource-performance relationship and bid accordingly; no prior knowledge of the workload characteristics is assumed.

Our evaluation shows that, using detailed simulations of a 64-core CMP configuration running a variety of multiprogrammed workloads, the proposed XChange mechanism improves system throughput (weighted speedup) by about 21% on average, and fairness (harmonic speedup) by about 24% on average, compared with equal-share on-chip cache and power distribution. On both metrics, that is at least about twice as much improvement over equal-share as a state-of-the-art centralized allocation scheme [9]. Furthermore, our results show that XChange is significantly more scalable than the state-of-the-art centralized allocation scheme we compare against: less than 0.5% overhead on a 5-million-cycle allocation interval (approx. 1 ms) to reach an allo-

cation decision, for CMP sizes anywhere from four to 128 cores. In contrast, the state-of-the-art centralized scheme we compare against takes over 30% of the allocation interval to converge under a 64-core CMP, and it exceeds the entire interval beyond 100 cores.

Although inspired by theoretical studies [16, 30, 46, 49], XChange is nevertheless heuristic by design. We present a comparison against a recently proposed, formally provable market-based resource allocation mechanism [48], and show that our heuristic approach delivers superior throughput across the board for the configurations and workloads studied.

This paper is organized as follows: Section 2 motivates our approach in contrast to existing art. Section 3 describes the general market framework that XChange is based on. Section 4 and Section 5 present the implementation of our proposed mechanism. Section 7 evaluates our proposal. Section 8 validates our model. Section 9 shows the scalability of our mechanism.

2. MOTIVATION OF OUR APPROACH

In the context of resource allocation of CMPs, researchers have shown that using fine-grained management of the available resources to provide optimized utilization is highly desirable as well as practical. Sanchez and Kozyrakis, for example, show that fine-grained shared-cache cache partitioning is feasible in a large-scale CMP system [36], yielding greatly improved utilization. Similarly, multiple power-oriented studies [6, 15, 23] show that fine-grained, per-core DVFS regulation can greatly improve a CMP's energy efficiency. Intel has recently deployed a low-cost, fully-integrated voltage regulator in Haswell [19], and other researchers are making significant advances in supporting per-core DVFS [21, 38]. The obvious downside of fine-grained resource allocation in large-scale CMPs is that the number of potential operating points can be large, making it more time-consuming to search for optimal allocation points.

When it comes to multi-resource allocation, uncoordinated solutions have been shown to be inefficient—even inferior to static equal-share partitioning—, due to their inability to model the interactions among resources [3]. A few solutions have been proposed to address the fine-grained multi-resource allocation problem. Approaches to estimate an allocation's performance have evolved, from simple trial runs [1, 10, 24], to behavior modeling based on artificial neural networks [3], or analytical models [9]. In all these cases, the solutions are primarily centralized mechanisms that seek to optimize system throughput by essentially exploring the resource allocation space sequentially. Unfortunately, coordinated multi-resource allocation dramatically increases the size of the this allocation space. As we will show in Section 9, centralized approaches are likely to be unfeasible for large-scale CMPs, as they may take too long to discover an optimized operating point that can be exploited effectively. Moreover, many of these techniques focus on throughput, with less concern for fairness.

Our proposed XChange solution tackles scalability by adopting a market-based approach. In a market-based approach, participants seek to optimize their resource assignment largely independently of each other, and participants' demands are reconciled through a pricing mechanism. Under relatively weak conditions (e.g., resources are priced equally for all participants at each point in time), such competitive markets can converge iteratively to a Pareto-efficient equilibrium (i.e., no further trading is mutually beneficial) [27]. These two properties, namely largely distributed operation and Pareto-efficient equilibrium, make a market approach potentially attractive in our context.

In the computer systems domain, Sutherland is believed to be the first one who created a market to manage Harvard University's computing resources [45]. Since then, a number of proposals have been put forward to allocate resources in distributed systems and data centers [8, 16, 18, 33]. Recently, Guevara et al. [18] use a market model to study the optimal configuration of heterogeneous data centers. They employ a "static market" view to allocate a single resource (compute service units): Users volunteer the amount of money each is willing to pay as a function of allocated service units. The central market then allocates the available computing resources so that monetary profit is maximized. This static view of a market is not useful in our context: To accomplish efficient multi-resource allocation, users should be able to adjust their bids dynamically in response to the perceived global resource contention—what is called the "price discovery" process. For example, a user can lower its bid for what turns out to be a highly contended resource (e.g., cache space) and bet on a different resource more heavily (e.g., power budget), if it concludes from supply-demand dynamics that it will get a better "bang for the buck."

Overall throughput is not the only concern to resource allocators: A measure of fairness is also highly desirable (e.g., to provide QoS). A recent proposal by Zahedi and Lee [48] applies an "elasticity-proportional" (EP) CMP resource allocation mechanism to accomplish game-theoretic fairness. Users' true resource utility is profiled, and the resulting profiles are curve-fitted to a log-linear function. The EP allocation mechanism uses these curve-fitted utility functions to provide an allocation with strong game-theoretic fairness guarantees, such as sharing incentives, envy freedom, and Pareto efficiency. However, guaranteeing game-theoretic fairness comes at a cost in system performance, and Zahedi and Lee's results indeed show that a fundamental trade-off exists between EP's game-theoretic fairness and achievable system throughput.

Our approach distances itself from pursuing provable game-theoretic guarantees, instead focusing on heuristics that can be practical and yield satisfactory levels of both throughput and fairness. We do not confine user behavior to a curved-fitted model, and hypothesize that a heuristic-based approximation of utility in the CMP, coupled with a fail-safe mechanism for outliers, should be sufficient to provide good outcomes. Intuitively, this is based in part on the fact that resource utility in CMPs, even if nonlinear, is monotonic (i.e., more of a resource yields equal or greater benefit)—a property present in many problems for which market-based solutions have been successful.[1] We measure throughput and fairness using metrics more conventionally found in the computer architecture community, namely weighted speedups and harmonic speedups.

Another limitation of Zahedi and Lee's approach [48] is that, although there seems to be no fundamental reason

[1] Technically, it is possible that some resources may exhibit some kind of Bélády's anomaly, where a slightly increased resource allocation actually hurts performance in certain cases. We did not find this to be an issue in our experiments.

why the utility profiles could not be derived online somehow, its evaluation is based on profiles obtained offline and a priori. While it may be possible in data centers to profile an application before dispatching it [11, 26], we believe this assumption is unrealistic for general CMP-based systems. XChange approximates resource utilities dynamically at run-time—no prior knowledge of the workload's behavior is necessary.

3. MARKET-BASED FRAMEWORK

Proper multi-resource allocation for CMPs presents the challenge of optimizing and balancing two system objectives, system throughput and fairness, as well as dealing with an allocation space which grows rapidly with the number of cores. To be truly practical, it also needs to be capable of building a resource-performance model dynamically at run-time, without the assistance of profiling or other sort of prior knowledge.

In this section, we describe the general market framework XChange is based on. We define the agents in the market as the applications running on the CMP cores. We consider the shared resources to be the chip's power budget and the last-level cache space. We regard the scheduling question of what apps to run on the available cores as orthogonal to our objective—after all, an agent would not spend any "money" on resources if it didn't get to run.

3.1 Overview

XChange operates as a market, where each processor in the CMP can "purchase" shared resources from the system. The pricing mechanism plays a central role in the market: it conveys supply and demand information, reflects the true value of the resources, and ultimately determines who gets how much of each resource [27].

We adopt the price-taking mechanism proposed by Kelly [22]: Assume R_j represents the total amount of resource j available, and b_{ij} is the amount of money agent i bids for resource j. Then p_j, the price of resource j, is computed as the total amount of money bid by all the agents on that resource, divided by the number of resource units available:

$$p_j = \frac{\sum_i b_{ij}}{R_j} \qquad (1)$$

The resources are then distributed proportionally to the bids each agent submits:

$$r_{ij} = \frac{b_{ij}}{p_j} \qquad (2)$$

Here r_{ij} is the amount of resource j allocated to agent i. Note that, because of the price-taking formulation, no part of the resource is left unallocated.

At each point in time, because resource prices are readily available to agents, the agents know exactly how much of a resource they would get given the bids they place for it, and therefore, these selfish agents are able to bid optimally to maximize their own utilities. During the bidding process, the prices will fluctuate. When prices become stable because agents have no incentive to change their bids to improve their utilities, the market has converged. This results in a *Pareto-efficient* resource allocation.

In addition, to ensure market fairness, each agent is assigned a finite budget, and the total amount of its bids cannot exceed that budget. We also do not allow agents to save money: any unused budget is forfeited if the agents do not use it. This is simpler than a situation where agents are allowed to save money to later try to monopolize all the resources, which probably hurts both convergence speed and fairness.

The entire bidding process is described as follows:

1. Initially, each agent builds its local utility function—i.e., its resource-performance relationship model. It is also assigned a budget to buy resources. Meanwhile, a global resource arbiter posts the initial prices for all resources to all agents. Under such prices, each agent places bids to buy these resources. These bids are such that they maximize the agent's local utility.

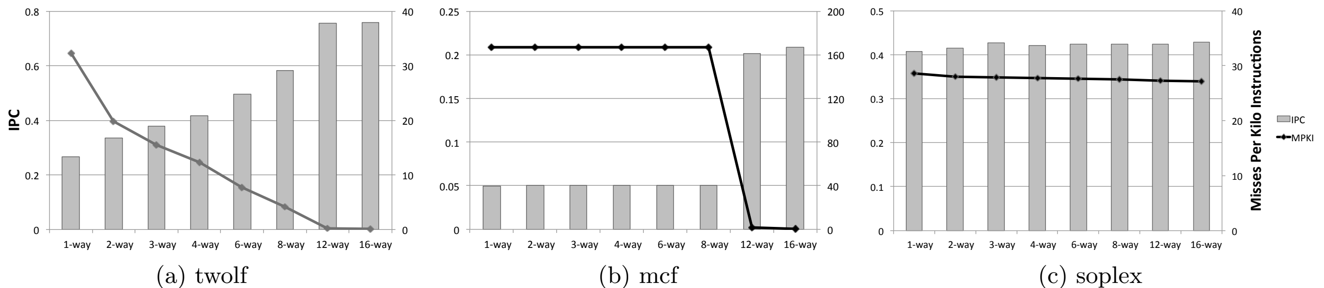
2. After all agents have placed their bids, a global resource arbiter collects the bids, and adjusts the prices of the resources based on Equation 1. The price of highly sought-after resources will be increased, and the price of unpopular resources will be lowered to promote sales. This is a quick process.

3. The resource arbiter posts the updated prices to all agents, who then bid again under the new pricing. This process repeats itself until the market converges—i.e., the price remains stable across iterations (within 1%), and the agents have no incentive to change their bids to improve their local utility. (We discuss more about convergence criteria in Section 4.3.2.) Finally, the resources are allocated as shown in Equation 2.

Prices play a key role here, as a reflection of overall system demand vs. supply. In other global optimization mechanisms for CMPs, only the *marginal* utility of each resource (i.e., the preference for that resource) is considered by the agents [9], regardless of whether it is highly contended or not. In our market, for example, if the price for resource A is high due to demand, an agent will start bidding more money on a cheaper resource B, *even though its marginal utility for resource A may be higher*, in an attempt to maximize its utility given the supply-demand circumstances.

Another major advantage of this process is that it is mostly done in a decentralized manner. Indeed, a key aspect of our market framework is that it takes advantage of individual wisdom: It allows the agents in the market to submit bids to maximize their local utility under the current resource prices, rather than submitting their utility function to a centralized entity that then performs a global search. Compared to prior centralized schemes proposed [3, 9], this process delegates the search effort to each individual agent. The only centralized work done in the system is the pricing mechanism shown in Equation 1, which is fairly simple and can be done efficiently. In addition, the overhead of collecting bids and posting prices is small.

One other interesting aspect of this market-based approach is that the trade-off between system throughput and fairness can be adjusted by assigning different budgets to different agents: Intuitively, if the system prefers higher throughput, it opts to assign higher budget to the agent with higher marginal utility; if the system prefers fair allocation across agents, it opts to assign equal budget. We discuss this issue further later in Section 4.3.3.



(a) twolf (b) mcf (c) soplex

Figure 1: IPC and cache miss rate under different cache allocation, running at highest possible frequency. The x axis is the number of cache ways enabled. Section 6 describes our experimental setup.

3.1.1 *Challenges of building the utility model*

The market model requires each agent to construct an accurate relationship between performance and resources—i.e., its local utility model. One possible solution could be to ask programmers or users to provide some "hint" to the on-chip agent about the dynamic behavior of the application. However, in general, programmers and users may provide the wrong incentive, primarily because they may be unaware of the hardware details. Profiling is an option, but this may not be feasible in practice, and in any case, the applications will be running in a different environment when it matters: different architectural configuration, competition with applications with different characteristics, etc. Notwithstanding these options, we propose to design an intelligent run-time monitoring mechanism, whose goal is to determine each agent's local utility model dynamically (and concurrently).

4. MECHANISM: MARKET PARTICIPANTS

In this section, we describe how each individual agent dynamically models its relationship between performance and resources, and its bidding strategy in reaction to the resource price under our market framework.

All other things being equal, a simpler model is usually preferable. Here we briefly discuss a linear model to provide an intuition of how market participants operate in general. As we will see shortly, when managing CMP resources the reality is not so simple (but it is manageable).

$$u_i = \sum_j w_{ij} \times r_{ij} \quad (3)$$

where w_{ij} represents agent i's marginal utility for resource j, r_{ij} represents the allocation of resource j to agent i, and u_i represents the overall utility of the allocation of resources for agent i. This linear utility function, where each agent's marginal utility for each resource is constant under all circumstances, is quite simple, and many bidding strategies have been proposed in the literature. For example, PR-dynamics, which is a bidding strategy based on the linear utility model, is theoretically proven by Zhang to guarantee fast market convergence, Pareto efficiency, and also game-theoretic fairness [49].

When we attempted to use a linear market by curve-fitting the utility functions of each application, and conducted a PR-dynamics-like market, the results were poor. In order to examine why the linear utility model is a poor fit to our specific CMP resource allocation problem, we profile a few applications with varying cache capacity. (See Section 6 for details on the experimental setup.) Figure 1 shows the L2 cache miss rate (MPKI) of three representative applications under different cache way allocation, and their corresponding IPC. All the applications run at the same frequency. We find that the cache-performance behavior of *soplex* and *twolf* can fit into the linear model pretty well: *soplex* doesn't benefit from more L2 (flat curve), and *twolf*'s IPC increases almost linearly with more cache ways.

However, *mcf* shows a step function in IPC and cache capacity: once it secures 12 ways (1.5MB), its working set can fit into the cache, and its miss rate drops to almost zero, showing a sudden 200% performance increase. Such IPC-cache utility curve does not fit a linear utility curve well—in fact, it is not even convex. The existing literature does not provide easy game-theoretic guarantees for agents that behave like *mcf*. We empirically observed that some other applications have similar behavior (admittedly, *mcf* is a bit extreme).

In XChange, we choose to abstain altogether from pursuing approaches with strong game-theory guarantees. Our approach is inspired by the fact that the First Welfare Theorem has relatively weak requirements to guarantee that any market equilibrium is Pareto-efficient: XChange is a market where agents are price-takers by design (i.e., they must accept the prices imposed by the market at each point in time); agents in XChange exhibit monotonic utility (i.e., more of a resource is better) by the nature of our problem; and agents in XChange always put forward their best bid (the one that they believe maximizes their utility given the current prices).

In the rest of this section we describe how we model XChange's utility function, in part by borrowing and combining successful hardware estimation mechanisms from the existing literature. As our evaluation will show, the model yields very good results for the configurations and workloads studied (Section 7).[2]

4.1 Utility Model

Existing literature frequently characterizes workload behavior by dividing its total execution time into memory phase (core stalled waiting for memory) and compute phase. In XChange, we borrow this simple compute-vs.-memory classification to characterize the impact of each of the shared resources on application behavior. Cache and off-chip band-

[2] The First Welfare Theorem states *sufficient* conditions for Pareto-efficiency under any market equilibrium. Thus, even market equilibria that do not strictly abide by such conditions may in principle be Pareto- or quasi-Pareto-efficient.

width are mostly related to the length of the memory phase: a larger L2 allocation will lower the cache miss rate, while more memory bandwidth will mitigate the penalty of cache misses. At the same time, a higher power budget allocation will allow a core to run at a higher frequency, and thus the compute phase will be scaled down proportionally [6, 28].

We define the agent's utility as the workload's execution time, i.e., the sum of its compute and memory phase, measured in cycles at nominal frequency (4GHz in our setup). We approximate compute and memory phases as being relatively independent—e.g., changing the core's power allocation should not affect the wait in the memory system due to cache misses and bandwidth traffic. This is of course a simplification: a lower clock frequency at the core, for example, *will* make the core issue memory requests at a slower speed, and thus affect the effective memory level parallelism, and the length of memory phase. However, it allows for simpler and faster models that, as our results will show, do deliver solid gains.

4.1.1 Cache Utility

We now describe how we derive our utility model for shared cache allocation, by combining two existing performance estimation mechanisms. Miftakhutdinov et al. develop a model to estimate the execution time of a program's memory phase [28]. In the model, a per-core memory critical path counter CP_{global} is maintained. When a memory request leaves the core and gets into the last-level cache, it copies CP_{global} to the its own counter CP_{local}. After Δt cycles, the memory request is served, and the critical path counter is set as $CP_{\text{global}} = \max(CP_{\text{global}}, CP_{\text{local}} + \Delta t)$. The value of counter CP_{global} reflects the length of the memory phase. (More details can be found in that paper.)

Unfortunately, this scheme is only able to estimate the length of memory phase under the current cache and bandwidth allocations. To estimate the cache's marginal utility, we need to be able to calculate the effect that a change in cache allocation has on the memory phase. Qureshi and Patt's UMON sampled cache tag array [32] can be used to predict the cache miss rate under all possible cache allocations, although is not able to directly predict the length of memory phase.

Thus, we extend the technique developed by Miftakhutdinov et al. by incorporating UMON. The simplifying assumption we make is that memory-level parallelism (MLP) doesn't change with different cache allocations, and therefore can be computed by dividing the aggregate service time by the length of the memory phase:

$$MLP = \frac{N_h \times t_h + N_m \times t_m}{CP_{\text{global}}} \tag{4}$$

where N_h and N_m are the number of hits and misses under the current cache allocation, respectively, and t_h and t_m are the hit and miss latencies, respectively.

In order to predict the length of memory phase under j cache ways, $CP_{\text{global}}(j)$, we compute the aggregate memory service time without MLP, by using the prediction of the number of hits and misses, $N_h(j)$ and $N_m(j)$ respectively, from UMON.[3] With MLP, the length of memory phase under j cache ways can be computed as:

[3] UMON with dynamic set sampling (DSS) is only able to predict miss rate, but we can multiply miss rate by the total number of cache accesses to obtain the number of hits and misses.

$$CP_{\text{global}}(j) = \frac{N_h(j) \times t_h + N_m(j) \times t_m}{MLP} \tag{5}$$

Therefore, assuming an agent is allocated i ways of cache, if it is given one less cache way, the increase in its execution time can be computed as:

$$MU_{\text{cache}}(i) = CP_{\text{global}}(i-1) - CP_{\text{global}}(i) \tag{6}$$

We define $MU_{\text{cache}}(i)$ as the agent's marginal utility for cache at i ways. (Note that lower CP_{global} is better, thus the order of the operands.)

4.1.2 Power Utility

The agent's marginal utility for power can be modeled based on the fact that the length of the compute phase tends to scale linearly with the processor frequency. By reading the appropriate hardware performance counters, the agent can collect the statistics from the last interval: length of compute phase t_{exe}, average operating frequency f_0, energy consumption E_0, and operating voltage V_0. Then the length of compute phase $t_{\text{exe}}(f)$ under new frequency f is:

$$t_{\text{exe}}(f) = t_{\text{exe}} \times \frac{f_0}{f} \tag{7}$$

The power is estimated as follows:

$$P(f) = \frac{\frac{E_0}{V_0^2} \times V_f^2}{t_{\text{exe}}(f) + t_{\text{mem}}} \tag{8}$$

Here, V_f is the voltage under new frequency f, and t_{mem} is the length of memory phase under current cache partition.

Therefore, assume an agent is operating at frequency f, we define its marginal utility for power as follows:

$$MU_{\text{power}}(f + \Delta f) = \frac{t_{\text{exe}}(f) - t_{\text{exe}}(f + \Delta f)}{P(f + \Delta f) - P(f)} \tag{9}$$

where Δf is the frequency increment of one DVFS step.

4.1.3 Bandwidth Utility

The marginal utility for memory bandwidth could be derived similarly, by taking effective memory latency into account when computing the length of memory phase; however in this paper for simplicity we assume an equal-share distribution across cores. This allows us to compare in the evaluation directly against state-of-the-art schemes for multi-resource allocation, which also allocate shared cache and power budget simultaneously [9]. Note that other resources, such as on-chip network bandwidth, can also be plugged into our utility model, as long as their resource-performance relationship can be accurately modeled.

4.2 Bidding Strategy

Agents in XChange conduct a simple local hill-climbing algorithm to find their optimal bids. Because the agent is working locally and independently, the complexity of this local hill-climbing does not increase with the number of cores. Notice that hill-climbing cannot generally guarantee the optimality of the solution, and thus the sufficient conditions of the First Welfare Theorem cannot be formally guaranteed. Nevertheless, again our experiments show that the bids produced in this way are of good quality.

We have established earlier in the paper that cache utility is generally not convex. This may cause hill-climbing to get stuck at a local optimum. For example, as is shown in Figure 1b, *mcf*'s marginal utility on allocating more cache ways is zero almost everywhere except for one point (8 to 12 ways). If the hill-climbing search starts by purchasing one cache way and the power consumption of the minimum possible frequency (800 MHz), it is almost guaranteed that the agent will bid heavily on power, because the marginal utility on cache is virtually zero in that region.

On the other hand, there is generally no such "knee" in the performance response to frequency (and thus power) changes; plus, its marginal utility diminishes as frequency increases, because power scales cubically with frequency, while compute time scales linearly (see Equation 9). These two "opposing" but otherwise monotonic behaviors enable us to design a "guided" hill-climbing search, which starts searching from the maximum affordable cache. Thus, our proposed local hill-climbing algorithm works as follows:

1. The price of resources is broadcast to the agents.
2. Each agent starts by purchasing its bare-minimum power (assuming the core is operating at 800 MHz), and leaves all the remaining budget to cache.
3. The agent decreases its cache bid by one way, and uses the saved money to buy extra power. By comparing the marginal utilities, the agent can decide whether the trade is worthwhile. If so, it accepts the trade and this step is repeated; otherwise, the agent denies the trade, and it submits the current bids to the market.

This algorithm should deal with *mcf*'s "step-like" cache behavior very well. In Figure 1b, the step of *mcf* is at 12 ways. Our hill-climbing starts from the maximum affordable cache. Suppose *mcf* can at most buy 10 cache ways because it is highly contended; then the algorithm will eventually end up trading 9 cache ways for power, and will never really worry about climbing up to 12 cache ways, simply because it is not affordable. On the contrary, if *mcf* can afford more than 12 cache ways, it will iterate as described above to decide, at each point, whether cache or power is more beneficial.

Our *guided* hill-climbing approach is efficient, because it walks through the search space linearly. However, this is true because only one resource in the system has a non-convex utility (the cache). In a more general case where multiple resources are not convex, more sophisticated algorithms such as Qureshi and Patt's Lookahead [32] would probably be needed.

4.3 Design Issues

In this section, we discuss three practical issues that must be addressed: bankruptcy, market convergence, and wealth redistribution.

4.3.1 Bankruptcy

Sometimes, the price for power can be so high in the market that an agent cannot even afford to buy the minimum power to operate at 800 MHz. In such an event, the agent will file for "bankruptcy." The agent will be excluded from the bidding process, and it will be allocated the bare minimum: one way of the cache, and allowed to operate at 800 MHz for the next interval. The other agents will bid for the remaining resources. The bankrupt agent will re-join the bidding process at the next interval.

4.3.2 Convergence

Many practical studies show that a market similar to ours is likely to operate quite well [16, 40]. Still, we do experimentally observe some circumstances where prices continue to oscillate by more than 1% (our convergence criterion). We could consider a market with a more relaxed convergence threshold, e.g. 5% fluctuation. This may eliminate some of the non-convergent cases, and also reduce the number of bidding iterations. However, it may lead to a less optimal allocation. Other convergence criteria exist in the literature; for example, utility fluctuation [16]. However, we observed experimentally that there was no practical difference between this and our original criterion in terms of system throughput or convergence rate.

One reason for continued oscillation is that cache allocation is done at the granularity of cache ways, and thus the utility function is not continuous. Agents may be swinging between two neighboring cache ways across iterations, resulting in a non-trivial fluctuation in cache price.

There comment on two potential solutions to deal with this situation. First, we can introduce a *price-smoothing* mechanism, by incorporating the price in the last iteration (p_j^{last}):

$$p_j = \alpha \times p_j^{\text{last}} + (1 - \alpha) \times \frac{\sum_i b_{ij}}{R_j} \tag{10}$$

In this way, the history of the price is factored in, which helps agents to better understand the contention of the resource in the market. We empirically pick α to be 0.2, and we observe that it can greatly improve the market convergence rate.

Another option is to adopt a *price-anticipating* mechanism [16] instead of our *price-taking* approach. In this mechanism, although an agent does not know how others will change their bids, it realizes that the increase/decrease of its bids will change the resource prices, according to Equation 1. Therefore, during the local search for optimal bids, the agent will no longer consider the price to be a fixed number: it will factor in the impact of its changing bid on the resource price when it tries to trade one cache way for power.

In our experimental setup, both solutions show similar system throughput and convergence rate. Because price-anticipating agents increase the complexity when bidding, in the rest of the paper we adopt the *price-smoothing* technique.

In any case, if the market ultimately cannot converge after a while, we have to announce that our market fails to converge. In XChange, we set the cut-off threshold to be 30 iterations; if the prices still fluctuate by more than 1% at that point, we terminate the bidding process. In that case, resources are allocated as follows: first, each agent estimates its utility under the current resource prices and its bids; then, each agent estimates its utility under an equal-share allocation. If one of the agent prefers equal share, we enforce the system to fall back to equal-share allocation for all agents. Otherwise, the resources are allocated according to the agents' last bids. In this way, such a "fail-safe" mechanism virtually guarantees that the allocation decision is at least as good as equal-share. In fact, our experiments show that in most cases, agents prefer the market outcome

Component	Quantity	Width	Bits
UMON shadow tag	16×64	28	28,672
UMON hit counter	16	32	512
DL1 CP_{global} counter	1	32	32
Per request CP_{global} counter	16	32	512
Total			29,728

Table 1: Per-core hardware overhead of online performance modeling.

rather than equal-share. This is especially true if multiple equilibria exist, and the market is simply oscillating among them.

4.3.3 Wealth Redistribution

In our initial design, our market framework treats all agents equally, by assigning to them the same initial budget. However, in our experiments, we find that an equal budget constraint might not be efficient enough from both a system and a user perspective. For example, *libquantum* is easily satisfied with very few resources. Because it's a highly memory-intensive application, its core stalls most of the time. As a result, it can operate at 4 GHz while consuming very little power (much lower than an equal-share power allocation). In addition, its working set can never fit into the shared cache, and allocating more than one cache way to it does not bring any significant benefit. But because it has the same budget as the other agents, it is likely to disrupt the market by preventing the other agents from obtaining resources that would contribute to a higher system throughput.

We propose a simple heuristic, namely to assign to each agent a budget proportional to its "potential" in performance gains:

$$B \propto (1 - \frac{t_{min}}{t_{max}}) \quad (11)$$

where t_{min} is the estimated execution time when the application is running alone (and thus enjoys all the chip's power and the maximum number of cache ways that UMON is able to monitor—see Section 5), and t_{max} is the estimated execution time when the application is running with minimum resources (one cache way and the lowest possible frequency). These quantities are not measured, but rather computed using Equation 5 and Equation 7 at the beginning of each partition interval; therefore, they do not lead to additional overhead.

This wealth redistribution technique biases budgeting toward the applications that have higher potential. As we discuss later in Section 7, this result in higher overall throughput, at the expense of some fairness. In those circumstances where fairness among cores are highly preferred, this wealth redistribution mechanism can be easily turned off.

5. IMPLEMENTATION

We propose to implement XChange as a combination of hardware and software. The hardware is responsible for performance monitoring and modeling, and the software is responsible for conducting the market's bidding and pricing mechanism.

5.1 Hardware

Table 1 details the per-core hardware overhead of our online performance modeling mechanism. As is discussed in Section 4, XChange relies on UMON shadow tags to predict cache behavior of applications. We employ a dynamic set sampling (DSS) technique [32] and sample 64 out of 2k sets. We further limit the stack distance to 16, because we empirically observe that no application can afford more than $4\times$ its equal-share allocation. In addition, for the workloads we study, we observe that their marginal utility for cache is mostly zero beyond 16 ways. However, in more general cases, a deeper stack distance may be needed to more accurately characterize the workloads' cache behavior, at the cost of higher storage overhead.

On top of that, in order to track the length of critical memory path CP_{global}, the L1 data cache of each core requires a global critical path counter CP_{global}, and each memory request needs a counter to save a copy of CP_{global} (as a field of DL1's MSHR). Further, it needs each processor to keep track of how much dynamic energy it consumes in the past interval. Because modern processors already have this feature built in [35], we exclude this from our hardware overhead.

In all, the per-core hardware overhead of XChange amounts to about 3,700 bytes.

5.2 Software

We propose to implement XChange's bidding-pricing mechanism as a part of an OS kernel module. In some Linux-based SMP systems, all cores are simultaneously interrupted by an APIC timer every 1 ms to conduct a kernel statistics update routine. We propose to piggyback on this interrupt to incorporate our market mechanism. We assume a shared-memory model, and designate a master core to be responsible for collecting the bids (reading shared variables) and computing the price. The whole procedure works as follows:

1. Every 1 ms, after each core has finished its kernel update routine, the master core posts an initial price.

2. All the cores start to search for their optimal bids using the local hill-climbing technique explained in Section 4.2.

3. After a global barrier to ensure that all bids are computed, the master core collects the bids, and computes the price. If the prices do not change (within 1%) compared to the previous iteration, the market converges, and the resources are allocated using Equation 2. Otherwise, repeat Step 2.

Because we are using a shared-memory model for the market mechanism, no special hardware is needed for inter-core communication. The execution time overhead of this procedure is discussed in Section 9.

5.2.1 Priorities and Real-Time Issues

In a real system there may be applications with different priorities. High-priority applications probably expect to enjoy more CPU cycles and access to more of the on-chip resources. Our market framework can handle this by assigning higher budgets to these applications, and therefore increase their purchasing power. How exactly to calculate the appropriate budgets is left for future work.

Another issue that high-priority or real-time applications may face could be caused by the fact that our market framework involves all the cores for the bidding-pricing procedure. Although the overhead is small (Section 9), these types of

Table 2: System configuration.

Chip-Multiprocessor System Configuration	
Number of Cores	8 / 64
Power Budget	80W / 640W [a]
Shared L2 Cache Capacity	4MB / 32MB
Shared L2 Cache Associativity	32 / 256 ways [b]
Memory Controller	2 / 16 channels
Core Configuration	
Frequency	0.8 GHz - 4.0 GHz
Voltage	0.8V - 1.2 V
Fetch/Issue/Commit Width	4 / 4 / 4
Int/FP/Ld/St/Br Units	2 / 2 / 2 / 2 / 2
Int/FP Multipliers	1 / 1
Int/FP Issue Queue Size	32 / 32 entries
ROB (Reorder Buffer) Entries	128
Int/FP Registers	160 / 160
Ld/St Queue Entries	32 / 32
Max. Unresolved Branches	24
Branch Misprediction Penalty	9 cycles min.
Branch Predictor	Alpha 21264 (tournament)
RAS Entries	32
BTB Size	512 entries, direct-mapped
iL1/dL1 Size	32 kB
iL1/dL1 Block Size	32 B / 32 B
iL1/dL1 Round-Trip Latency	2 / 3 cycles (uncontended)
iL1/dL1 Ports	1 / 2
iL1/dL1 MSHR Entries	16 / 16
iL1/dL1 Associativity	direct-mapped / 4-way
Memory Disambiguation	Perfect

[a] We anticipate that the CMP systems with different number of cores will not be fabricated under the same technology. For simplicity, in our evaluation, we use a chip TDP of 10W per core. IBM's Power8 reportedly consumes twice as much per core (it has 12 cores).

[b] In the evaluation, we partition the shared last-level cache at the granularity of cache ways [32]. In an actual implementation, any of the fine-grained cache partition mechanisms proposed in the literature could be used (e.g., PriSM [25], Vantage [36]).

applications may be affected undesirably. One way to handle such cases may be to delegate the bidding on behalf of such time-sensitive applications on low-priority cores. Other solutions may be possible.

Yet another issue for real-time applications may be that the resulting resource allocation may be insufficient to meet hard deadlines. We propose address this by providing those applications with enough resources, and then excluding them from the market. (Their resource demands could be provided externally, or they could be estimated using the utility model we described in Section 4.1.)

6. EXPERIMENTAL METHODOLOGY

6.1 Architectural Model

We evaluate XChange using a heavily modified version of SESC [34]. The CMP configurations with 8 (small-scale) and 64 (large-scale) cores, and the out-of-order core parameters are shown in Table 2. We also faithfully model Micron's DDR3-1600 DRAM timing [20], shown in Table 3.

We use Wattch [4] and Cacti [37] to model the dynamic power consumption of the processors and memory system. The static power consumption is approximated as a fraction of the dynamic power, and this fraction ratio is exponentially dependent on the system temperature [7]. Intel has adopted a similar approach for its Sandy Bridge power management [35]. We rely on Hotspot [39] integrated with SESC to estimate the run-time temperature of our CMP system.

Our baseline CMP system is able to regulate three shared on-chip resources: L2 cache, off-chip memory bandwidth,

Table 3: DRAM parameters.

Micron DDR3-1600 DRAM	[20]
Transaction Queue	64 entries
Peak Data Rate	12.8 GB/s
DRAM Bus Frequency	800 MHz (DDR)
Number of Channels	2 / 16
DIMM Configuration	Dual rank
Number of Banks	8 per rank
Row Buffer Size	1 KB
Address Mapping	Page Interleaving
Row Policy	Open Page
Burst Length	8
tRCD	10 DRAM cycles
tCL	10 DRAM cycles
tWL	7 DRAM cycles
tCCD	4 DRAM cycles
tWTR	6 DRAM cycles
tWR	12 DRAM cycles
tRTP	6 DRAM cycles
tRP	10 DRAM cycles
tRRD	6 DRAM cycles
tRTRS	2 DRAM cycles
tRAS	28 DRAM cycles
tRC	38 DRAM cycles
Refresh Cycle	8,192 refresh commands every 64 ms
tRFC	128 DRAM cycles

and power budget. We distribute the power budget across the chip via per-core DVFS. When a processor exceeds its allocated power, it is forced to slow down until its power consumption drops within its share. We guarantee that each core receives at least one cache way; the remaining cache is distributed based on the resource allocation decision. Finally, we implement the Fair-Queue (FQ) memory scheduler proposed by Nesbit *et al.* [29] to regulate off-chip memory bandwidth. The service share rate will be designated at the memory controller by the resource allocator.

Workload Construction

We use a mix of 25 applications from SPEC2000 [41] and SPEC2006 [42] to evaluate our proposal. Each application is cross-compiled to a MIPS executable, using gcc 4.6.1 at -O2 optimization level. The bundles of applications are executed until every application has committed 200 million instructions (8 cores), or 50 million instructions (64 cores). When an application finishes, we stop measuring its performance, but continue executing and engaging it in global resource allocation, to ensure that it continues to exert pressure on the resources shared with other applications in the bundle. For each application, we use Simpoints [5] to pick the most representative program slice.

The multiprogrammed workloads we use are shown in Table 4. We classify the 25 applications into *Power-sensitive*, *Cache-sensitive*, and *Memory-sensitive* using profiling, and then create bundles that constitute a varied mix of applications in each category. When the number of cores exceeds the number of apps in a bundle, the bundle is replicated across the chip. For example, 8 copies of VCUGXNTZ will run to occupy all the cores in a 64-core CMP.

Our evaluation is based on multiprogrammed workloads because we anticipate to allocate resources at the granularity of applications. For multithreaded workloads, we can either treat each thread as an individual agent in the market, or combine all the threads of one app as one agent to bid and share the resources.[4]

[4] Skewing resources among threads in a multithreaded application (e.g., to alleviate synchronization imbalance) is beyond our scope, can be incorporated orthogonally to our approach, and has been studied elsewhere [2, 13].

Table 4: Multiprogrammed workloads, combining cache-, processor- and memory-sensitive applications.

Bundle	Applications	Type
ROAFLDGV	vpr - twolf - apsi - mcf	C^4
	milc - GemsFDTD - gromacs - vortex	M^2P^2
XNIBDWFA	soplex - libquantum - leslie3d - bwaves	M^4
	GemsFDTD - swim - mcf - apsi	M^2C^2
MHKULBFP	gamess - hmmer - sixtrack - wupwise	P^4
	milc - bwaves - mcf - ammp	M^4
FOARLNBW	mcf - twolf - apsi - vpr	C^4
	milc - libquantum - bwaves - swim	M^4
AZFPIBDW	apsi - bzip2 - mcf -ammp	C^4
	leslie3d - bwaves - GemsFDTD - swim	M^4
XIDWCHAF	soplex - leslie3d - GemsFDTD - swim	M^4
	calculix - hmmer - apsi - mcf	P^2C^2
NXLIGEOR	libquantum - soplex - milc leslie3d	M^4
	gromacs - h264ref - twolf - vpr	P^2C^2
LNIBDWOT	milc - libquantum - leslie3d - bwaves	M^4
	GemsFDTD - swim - twolf - art	M^2C^2
ROPAFZXN	vpr - twolf - ammp - apsi	C^4
	mcf - bzip2 - soplex - libquantum	C^2M^2
LXNIBDRO	milc - soplex - libquantum - leslie3d	M^4
	bwaves - GemsFDTD - vpr - twolf	M^2C^2
VCUGXNTZ	vortex - calculix - wupwise - gromacs	P^4
	soplex - libquantum - art - bzip2	M^2C^2

Performance Metrics

A key issue in resource allocation is the figure of merit. Eyerman and Eeckhout propose that two metrics be reported for a CMP system running multiprogrammed workloads: One to represent a system perspective, which cares about system throughput; and one to represent a user perspective, which cares about the average turnaround time of an individual job. The proposed metrics are weighted speedup and average slowdown of co-running applications (the reciprocal of harmonic speedup), respectively:

$$\text{Weighted Speedup} = \frac{1}{N}\sum_{i=1}^{N}\frac{IPC_i^{\text{shared}}}{IPC_i^{\text{alone}}} \quad (12)$$

$$\text{Harmonic Speedup} = \frac{N}{\sum_i \frac{IPC_i^{\text{alone}}}{IPC_i^{\text{shared}}}} \quad (13)$$

A system could achieve high throughput (i.e., weighted speedup) by starving one or two applications while benefiting all the others; however, system fairness (i.e., harmonic speedup) would suffer as a result, providing a bad experience to some users. A side-by-side comparison of weighted and harmonic speedups would expose this behavior. To isolate the fairness component, we also report separately the ratio between maximum and minimum slowdowns across the bundle [12]:

$$\text{Slowdown Ratio} = \frac{\max_i \frac{IPC_i^{\text{shared}}}{IPC_i^{\text{alone}}}}{\min_i \frac{IPC_i^{\text{shared}}}{IPC_i^{\text{alone}}}} \quad (14)$$

7. EVALUATION

Figure 2 reports system throughput (weighted speedup), slowdown ratio, and fairness (harmonic speedup) for an equal-share allocation (EqualShare), two competing mechanisms (GHC [9] and REF [48]), as well as XChange with (-WR) and without (-NoWR) wealth redistribution. System through-

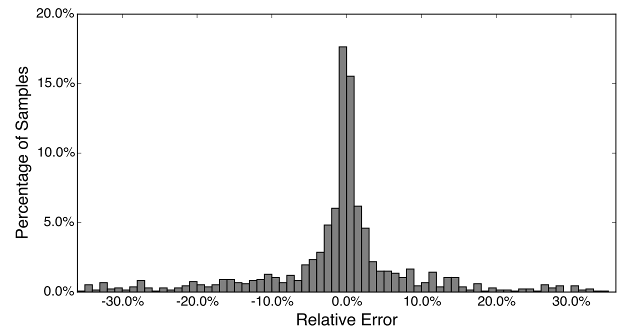


Figure 3: Accuracy of XChange in predicting the length of memory phase.

put and harmonic speedup results are normalized to an unmanaged allocation (Unmanaged). Unmanaged adopts LRU as the policy to manage shared cache, and full-chip DVFS rather than per-core DVFS, to guarantee that the chip's power consumption does not exceed its TDP.

The results are obtained by modeling the allocation algorithms faithfully, however the timing overhead of running these algorithms is set to zero in all cases. In the next section, we show that XChange's actual overhead is absolutely and relatively very low, and that it scales much better than GHC as the number cores/apps increases. Furthermore, the results for *REF* assume prior app profile knowledge [48]. Thus, any comparison with the competing mechanisms here tends to favor those and go against XChange.

7.1 XChange vs. Unmanaged

We first compare XChange against the Unmanaged baseline. Figure 2 shows that, on average, both XChange-NoWR and -WR improve system throughput significantly—by 13.62% (6.01%) and 18.30% (12.67%), respectively, for the 64 (8) CMP configuration. Looking at each individual bundle, we find that, although Unmanaged is almost universally inferior to XChange in terms of weighted speedup, it modestly outperforms XChange for ROPAFZXN in both the 8- and the 64-core configurations. We now look at this case a bit more closely.

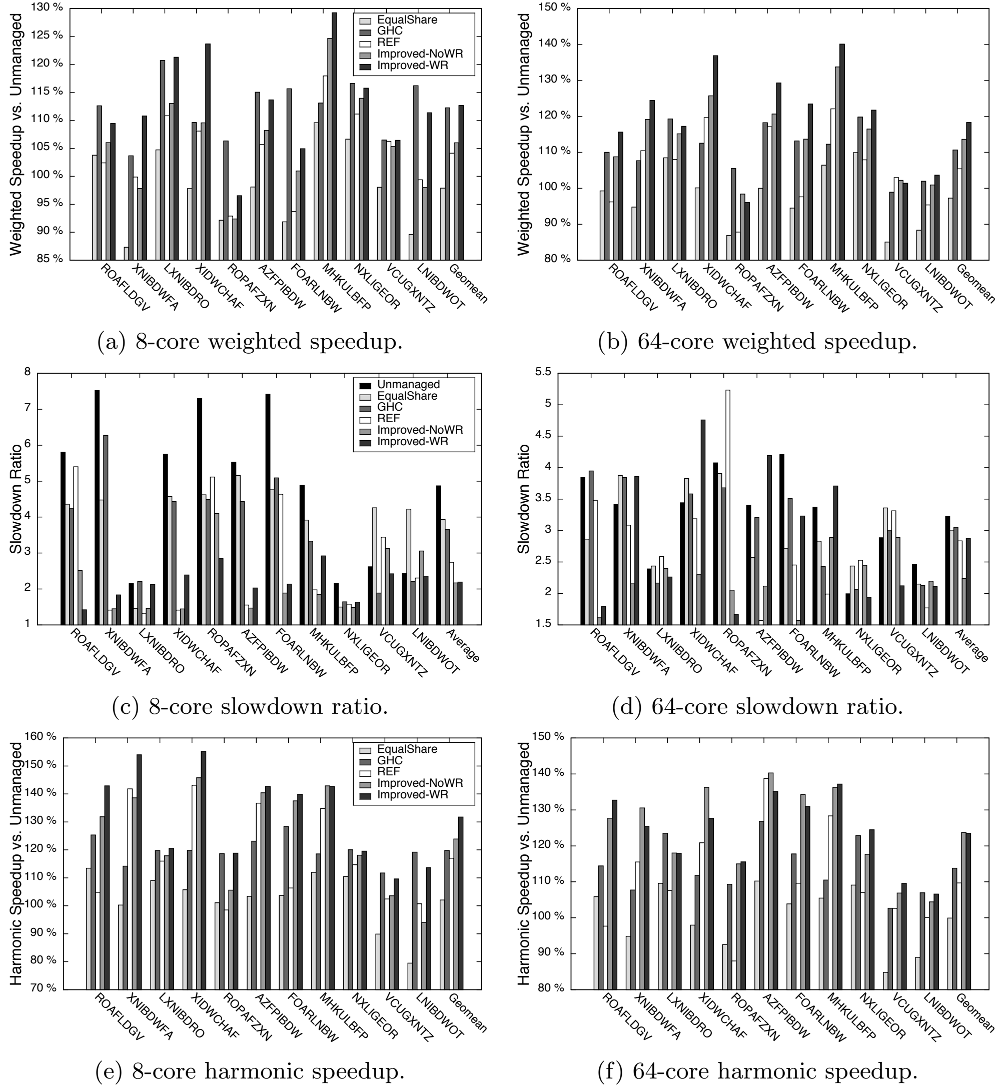
Bundle ROPAFZXN has six out of eight cache-sensitive applications. In Unmanaged, two of the cache-sensitive applications manage to hoard most of the cache space, letting the other four starve. The market-based approaches with built-in fairness (XChange but also REF and EqualShare) naturally do not exhibit this behavior.

On the other hand, the XChange configurations yield clearly superior slowdown ratio and harmonic speedup for that bundle over Unmanaged, in fact at a level that the other fairness-aware configurations fall well short of. On average across all bundles, and for the 8-core configuration, XChange-NoWR and -WR outperform Unmanaged in harmonic speedup (23.87% and 31.74%, respectively), and also slowdown ratio (2.17 and 2.19, respectively, vs. 4.87). The results are similar for larger 64-core configuration.

Overall, XChange outperforms Unmanaged almost universally in all three metrics.

7.2 XChange vs. EqualShare

We now compare XChange against the equal-share allo-



(a) 8-core weighted speedup.

(b) 64-core weighted speedup.

(c) 8-core slowdown ratio.

(d) 64-core slowdown ratio.

(e) 8-core harmonic speedup.

(f) 64-core harmonic speedup.

Figure 2: Comparison of system throughput (weighted speedup; higher is better), slowdown ratio (lower is better), and fairness (harmonic speedup; higher is better) among EqualShare, GHC, REF, XChange-NoWR, and XChange-WR, under different CMP configurations. System throughput and harmonic speedup results are normalized to Unmanaged.

cation (EqualShare). Figure 2 shows that, for the 64-core configuration, both XChange-NoWR and XChange-WR improve system throughput by 16.33% and 21.01% on average, respectively. In fact, XChange is superior in all the experiments. Results for the smaller 8-core configuration are similarly significant.

Looking a bit more closely at a representative bundle ROAFLDGV, we find that in XChange-NoWR, but more so in XChange-WR, *mcf* is able to stay atop the "cache utility step" and obtain 12 cache ways. In XChange-NoWR, *mcf* spends most of its budget to accomplish this, and as a result power is somewhat sacrificed, for an overall dampened performance gain. On the other hand, XChange-WR observes that *mcf*'s potential is high compared to the other apps, and consequently it assigns it a higher budget. Then, *mcf* can afford to "purchase" more power to run faster (Section 4.3.3), which results in higher speedups.

Notice that XChange-NoWR and -WR also yield superior fairness to EqualShare (2.24 and 2.88, respectively, vs. 3.0 average slowdown ratio). For example, *libquantum* is both cache- and power-insensitive, and thus its slowdown under any resource allocation is negligible. On the other hand, applications that are power-hungry (e.g., *calculix*) or cache-hungry (e.g., *art*, *mcf*) will experience a significant slowdown in EqualShare. The allocations by the XChange configurations are more balanced in that regard.

Overall, the combination of higher throughput and improved fairness makes XChange-NoWR and -WR significantly outperform EqualShare by 23.78% and 23.60%, respectively, in average harmonic speedup.

7.3 XChange vs. GHC, REF

Configuration *GHC* corresponds to Chen and John [9] with their online performance modeling, and *REF* corresponds to Zahedi and Lee with prior app profile knowledge, as evaluated in that work [48]. When compared against

GHC and REF, XChange-WR is superior in every metric, and the all-out winner. This holds for 64- as well as 8-core experiments. The harmonic speedup summarizes this well, with XChange-WR's 23.53% (31.74%) average easily outdoing GHC's 13.80% (19.80%) and REF's 9.70% (17.03%) in the 64 (8) setup.

When looking at REF more closely, we find that in many cases it is usually too biased toward maintaining game-theoretic fairness guarantees, and realized throughput gains suffer as a result. Although these may generally translate into a better user experience, our mechanism succeeds at making more aggressive decisions to maximize system throughput, while still striving for spreading fairly the impact across all applications' execution times.

7.4 Overall Effect of Wealth Redistribution

As discussed in Section 4.3.3, we introduce a wealth redistribution technique to further improve system throughput, possibly at some expense of fairness. Our simulations prove this intuition: XChange-NoWR achieves the best slowdown ratio (2.23 and 2.17 for 64- and 8-core, respectively) over all other techniques. In the meantime, XChange-WR achieves the best weighted speedup, and is consistently 5% better than XChange-NoWR. On the whole, XChange-WR slightly outperforms -NoWR in harmonic speedup.

Section 7.2 describes an example where *mcf* benefits by the budget redistribution. Let us briefly discuss another example, XIDWCHAF, in the 8-core configuration. In XChange-WR, the budget of cache- and power-sensitive apps such as *apsi* and *calculix* are offered higher budget than memory-bound apps, because their potential in deriving speedups from resources is higher. As a result, XChange-WR's weighted speedup over Unmanaged is about three times higher than XChange-NoWR's (23.68% vs. 9.56%, respectively), but on the other hand its slowdown ratio increases from 1.45 (XChange-NoWR) to 2.39.

8. FIRST-ORDER MODEL VALIDATION

Although we have shown significant improvements for both throughput and fairness, a measure of XChange's model accuracy is a useful insight. A key aspect of XChange's utility model is the estimation of the memory phase, which relies on the simplifying assumption that MLP remains unchanged across different cache allocations for any one application. To validate this memory phase estimation, we run each application alone with all possible cache allocations. Each run will give us the real length of memory phase under that specific cache capacity, and the estimates for length of all the others.

Figure 3 shows the accuracy of the estimation. The average error is 7.63%, indicating that our estimation is reasonably accurate. In general, we find that accuracy decreases when predicting for cache allocations that are more distant from the current allocation (e.g., predicting the length of memory phase under one cache way when the core currently owns eight cache ways).

We also find that another source of error is UMON: With very limited L2 allocated cache size, L1 cache lines will be more often invalidated due to replacements in the L2 cache. As a result, for some applications, the L1 miss rate will increase as L2's allocation decreases, which is not captured well by UMON.

# cores	8	32	64	128	256
GHC					
Cycles (K)	43	484	1,697	6,418	24,903
% interval	0.87%	9.69%	33.95%	128%	498%
XChange-WR					
Cycles (K)	9.47	12.49	15.89	22.64	52.70
% interval	0.19%	0.25%	0.32%	0.45%	1.05%

Table 5: Search overhead for GHC and XChange-WR. Interval is 5 million cycles.

9. SCALABILITY

Our simulations so far have excluded from all the configurations studied the overhead of searching through the resource allocation space. In this section, we study the scalability of XChange against GHC when that overhead is factored in. The hardware setup is as follows: an N-core CMP consumes $10N$ W of power and holds a $4N$-way, $0.5N$ MB L2 cache. Memory bandwidth is set to equal-share with FQ scheduling [29]. We limit the amount of cache that a single core can appropriate to 16 ways (2 MB) due to the UMON hardware overhead discussed in Section 5. The hardware configuration is the same as the simulation described in Section 6, and the synchronization/communication overhead across cores is included in all cases.

Recall from Section 5.2 that we anticipate the resource allocation mechanisms to be implemented in the kernel. We actually implement GHC and XChange as programs that we run on our simulation platform, and use the number of cycles each algorithm takes to converge as the metric to measure scalability.

As shown in Table 5, the total cycle count of GHC grows essentially quadratically. Recall that GHC is inherently sequential: A single core is responsible for the entire search. During the hill-climbing period, that core has to stop its normal execution to make the resource allocation decision on behalf of the entire CMP. With 64 cores on the chip, it would take that core 34% of a 5-million-cycle interval to come up an allocation decision. During that time, all the other cores would be running in a obsolete, probably suboptimal operating point. Note that, even in cases in which the performance of the old and new allocations for the interval were similar, the overall performance would be no better than the one reported earlier in the evaluation. For a CMP with more than 64 cores, it is simply unfeasible to apply GHC for the interval chosen.

In contrast, the XChange market-based mechanism comes to an allocation decision much more quickly. This is mainly for two reasons: (1) because most of the work is done concurrently across all cores; and (2) because the local allocation space each core needs to search is relatively small.

Table 5 shows the average cycle count for XChange-WR to converge, and the percentage of the partitioning interval every core will diverge from normal execution to compute allocation decision. For CMP systems with fewer than 128 cores, the system downtime of all market-based models is less than 0.5%. Above 128 cores, the cycle count begins to increase more or less linearly with the number of cores. This is because the master core needs to collect and sum up all the bids from the agents in the system to compute the resource price, and this centralized step starts to dominate the overall cycle count.

A potential way to alleviate this is to parallelize the price

computation, which is basically a reduction operation over the bids from the agents, into a tree fashion. Another option is to make the partition interval longer (also for GHC), but this may make the market too insensitive to application phase changes. We leave these and other possible options as future work.

10. RELATED WORK

Suh et al. [43] propose to distribute the L2 cache ways to minimize its overall miss rate. Qureshi and Patt [32] predict the marginal utility of additional cache ways for each application. Xie and Loh instead manipulate the cache insertion and promotion policy [47]. Vantage [36] and PriSM [25] propose fine-grained cache partitioning that scales well to large CMP systems.

Isci et al. propose a dynamic model which is able to predict the execution time and power consumption of a core under different operating frequencies. Based on this model, they design a power management technique which optimizes system throughput under a certain power budget [6]. Miftakhutdinov et al. improve upon that model by more accurately predicting the execution time, and show good energy savings under a given performance target [28].

However, uncoordinated resource allocation has been shown to be inefficient, because it is unable to deal with interactions among resources [3]. A few solutions have been proposed to address the fine-grained multi-resource allocation problem, primarily based on global optimization [1, 3, 10, 24]. XChange improves upon these works by applying a purely dynamic, largely distributed technique, which is able to deliver scalable system throughput and fairness in large CMP systems.

Petrica et al. propose a reconfigurable architecture that scales core resources down to efficiently utilize the limited available power [31]. Ghasemi et al. improve upon this by jointly scaling the core resources as well as the shared resources [17].

Resource allocation for multithreading applications is an interesting topic, because performance is not only limited by allocated resources, but also by the interactions among threads. Bhattacharjee and Martonosi propose a thread criticality predictor to predict the critical threads of an application, and giving more resources to them to rebalance computation [2]. Ebrahimi et al. adopt a similar technique by skewing memory bandwidth among threads [13]. Suleman et al. propose a feedback mechanism to dynamically figure out the most efficient number of threads an application should spawn to maximize resource utilization [44].

11. CONCLUSION

We have proposed XChange, a market-based mechanism to dynamically allocate multiple resources in CMPs. By formulating the CMP as a market, where each core pursues its own benefit, the system is able to maintain a good balance between system throughput and fairness. Our evaluation shows that, compared against an equal-share allocation, our market-based technique improves system throughput (weighted speedup) on average by 21%, and fairness (harmonic speedup) on average by 24% in a 64-core CMP system. Compared with a state-of-the-art centralized allocation scheme [9], that is at least about twice as much improvement over the equal-share allocation.

We have also shown that our market-based mechanism is largely distributed, where agents concurrently strive to maximize their individual utility. As a result, our approach converges significantly faster than the state-of-the-art centralized optimization technique we compare against.

Acknowledgments

We are grateful to the anonymous reviewers for their thoughtful feedback, which helped improve the paper. This work was supported in part by NSF award CCF-0720773, and by Intel's Science and Technology Center for Embedded Computing.

12. REFERENCES

[1] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Intl. Conf. on Computing Frontiers (CF)*, 2006.

[2] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Intl. Symp. on Computer Architecture (ISCA)*, 2009.

[3] R. Bitirgen, E. İpek, and J.F. Martínez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Intl. Symp. on Microarchitecture (MICRO)*, 2008.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Intl. Symp. on Computer Architecture (ISCA)*, 2000.

[5] B. Calder, T. Sherwood, E. Perelman, and G. Hamerley. Simpoint. http://www.cs.ucsd.edu/ calder/simpoint/, 2003.

[6] C. Isci, C.-Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Intl. Symp. on Microarchitecture (MICRO)*, 2006.

[7] P. Chaparro, J. González, and A. González. Thermal-effective clustered microarchitectures. In *Wkshp. on Temperature-Aware Computer Systems*, 2004.

[8] J.S. Chase, D.C. Anderson, P.N. Thakar, A.M. Vahdat, and R.P. Doyle. Managing energy and server resources in hosting centers. In *ACM Symp. on Operating Systems Principles (SOSP)*, 2001.

[9] J. Chen and L.K. John. Predictive coordination of multiple on-chip resources for chip multiprocessors. In *Intl. Conf. on Supercomputing (ICS)*, 2011.

[10] S. Choi and D. Yeung. Learning-based SMT processor resource distribution via hill-climbing. In *Intl. Symp. on Computer Architecture (ISCA)*, 2006.

[11] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[12] E. Ebrahimi, C.J. Lee, O. Mutlu, and Y.N. Patt. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[13] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C.J. Lee, J.A. Joao, O. Mutlu, and Y.N. Patt. Parallel application memory scheduling. In *Intl. Symp. on Microarchitecture (MICRO)*, 2011.

[14] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.

[15] S. Eyerman and L. Eeckhout. Fine-grained DVFS using on-chip regulators. *ACM Trans. on Architecture and Code Optimization (TACO)*, 8(1), 2011.

[16] M. Feldman, K. Lai, and L. Zhang. A price-anticipating

resource allocation mechanism for distributed shared clusters. In *Intl. Conf. on Electronic Commerce (EC)*, 2005.

[17] H.R. Ghasemi and N.S. Kim. RCS: runtime resource and core scaling for power-constrained multi-core processors. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[18] M. Guevara, B. Lubin, and B.C. Lee. Navigating heterogeneous processors with market mechanisms. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2013.

[19] P. Hammarlund, A.J. Martinez, A.A. Bajwa, D.L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R.B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The fourth-generation Intel core processor. *IEEE Micro*, 34(2):6–20, 2014.

[20] 2Gb DDR3 SDRAM component data sheet: MT41J256M8. http://www.micron.com/parts/dram/ddr3-sdram/mt41j256m8da-125, July 2012.

[21] R. Jevtic, H.-P. Le, M. Blagojevic, S. Bailey, K. Asanovic, E. Alon, and B. Nikolic. Per-core DVFS with switched-capacitor converters for energy efficiency in manycore processors. *IEEE Trans. on Very Large Scale Integration (TVLSI) Systems*, 2014.

[22] F. Kelly. Charging and rate control for elastic traffic. *European Trans. on Telecommunications*, 8(1), 1997.

[23] W. Kim, M.S. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2008.

[24] 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 *Intl. Symp. on Microarchitecture (MICRO)*, 2003.

[25] R. Manikantan, K. Rajan, and R. Govindarajan. Probabilistic shared cache management (PriSM). In *Intl. Symp. on Computer Architecture (ISCA)*, 2012.

[26] J. Mars, L. Tang, R. Hundt, K. Skadron, and M.L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Intl. Symp. on Microarchitecture (MICRO)*, 2011.

[27] A. Mas-Colell, M.D. Whinston, J.R. Green. *Microeconomic Theory*. Oxford University Press New York, 1995.

[28] R. Miftakhutdinov, E. Ebrahimi, and Y.N. Patt. Predicting performance impact of DVFS for realistic memory systems. In *Intl. Symp. on Microarchitecture (MICRO)*, 2012.

[29] K.J. Nesbit, N. Aggarwal, J. Laudon, and J.E. Smith. Fair queuing memory systems. In *Intl. Symp. on Microarchitecture (MICRO)*, 2006.

[30] N. Nisan, T. Roughgarden, É. Tardos, and V.V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, 2007.

[31] P. Petrica, A.M. Izraelevitz, D.H. Albonesi, and C.A. Shoemaker. Flicker: A dynamically adaptive architecture for power limited multicore systems. In *Intl. Symp. on Computer Architecture (ISCA)*, 2013.

[32] M.K. Qureshi and Y.N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Intl. Symp. on Microarchitecture (MICRO)*, 2006.

[33] O. Regev and N. Nisan. The POPCORN market. Online markets for computational resources. *Decision Support Systems*, 28(1):177–189, 2000.

[34] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator. http://sesc.sourceforge.net, 2005.

[35] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro*, 32(2):0020–27, 2012.

[36] D. Sanchez and C. Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *Intl. Symp. on Computer Architecture (ISCA)*, 2011.

[37] P. Shivakumar and N.P. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Tech. Rep., HP Western Research Labs, 2001.

[38] A.A. Sinkar, H.R. Ghasemi, M.J. Schulte, U.R. Karpuzcu, and N.S. Kim. Low-cost per-core voltage domain support for power-constrained high-performance processors. *IEEE Trans. on Very Large Scale Integration (TVLSI) Systems*, 22(4):747–758, 2014.

[39] K. Skadron, M.R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. on Architecture and Code Optimization (TACO)*, 1(1):94–125, 2004.

[40] A. Smith. *An Inquiry into the Nature and Causes of the Wealth of Nations*. A. and C. Black, 1863.

[41] Standard Performance Evaluation Corporation. SPEC CPU2000. http://www.spec.org/cpu2000/, 2000.

[42] Standard Performance Evaluation Corporation. SPEC CPU2006. http://www.spec.org/cpu2006/, 2006.

[43] G.E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2002.

[44] M.A. Suleman, M.K. Qureshi, and Y.N. Patt. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[45] I.E. Sutherland. A future market in computer time. *Comm. of the ACM*, 11, 1968.

[46] F. Wu and L. Zhang. Proportional response dynamics leads to market equilibrium. In *Intl. Symp. on Theory of Computing*, 2007.

[47] Y. Xie and G.H. Loh. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Intl. Symp. on Computer Architecture (ISCA)*, 2009.

[48] S.M. Zahedi and B.C. Lee. REF: Resource elasticity fairness with sharing incentives for multiprocessors. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[49] L. Zhang. Proportional response dynamics in the Fisher market. *Theoretical Computer Science*, 412(24), 2011.