Open access • Proceedings Article • DOI:10.1109/SAMOS.2013.6621127

TimeCube: A manycore embedded processor with interference-agnostic progress tracking — Source link

Anshuman Gupta, Jack Sampson, Michael Taylor

Institutions: University of California, San Diego

Published on: 15 Jul 2013 - International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation

Topics: Performance per watt and Resource allocation (computer)

TimeCube: A Manycore Embedded Processor with Interference-Agnostic Progress Tracking

Anshuman Gupta Jack Sampson Michael Bedford Taylor
Computer Science and Engineering
University of California, San Diego
California, USA

Abstract— **Recently introduced processors such as Tilera's Tile Gx100 and Intel's 48-core SCC have delivered on the promise of high performance per watt in manycore processors, making these architectures ostensibly as attractive for low-power embedded processors as for cloud services. However, these architectures space-multiplex the microarchitectural resources between many threads to increase utilization, which leads to potentially large and varying levels of interference. This decorrelates CPU-time from actual application progress and decreases the ability of traditional software to accurately track and finely control application progress, hindering the adoption of manycore processors in embedded computing.**

In this paper we propose *Progress Time* as the counterpart of CPU-time in space-multiplexed systems and show how it can be used to track application progress. We also introduce TimeCube, a manycore embedded processor that uses dynamic execution isolation and shadow performance modeling to provide an accurate online measurement of each application's Progress Time. Our evaluation shows that a 32-core TimeCube processor can track application progress with less than 1% error even in the presence of a 6× average worst-case slowdown. TimeCube also uses Progress Times to perform online architectural resource management that leads to a 36% improvement in throughput compared to existing microarchitectural resource allocation schemes. Overall, the results argue for adding the requisite microarchitectural structures to support Progress Time in manycore chips for embedded systems.

I. Introduction

Multicore processors have already become ubiquitous in some embedded domains, such as smart phones. In these, as in other domains utilizing multiprocessors, there is a trend toward greater concurrency that will soon move us from an era of multicore designs into an era of manycore designs. Manycore processors are especially attractive for embedded applications because they optimize energy per operation for high compute workloads as demonstrated by recent manycore offerings, such as Tile Gx100 [1] and Intel's SCC [2].

The aforementioned processors, being originally designed for cloud applications, needed to meet a set of key demands, shown in Table I. However, finding the right balance in managing architectural resources for an embedded system features a parallel set of demands. Thus, similar architectures are likely to be appropriate for both the cloud and embedded spaces, and similar mechanisms may be employed to address any issues limiting their adoption; Facebook, for instance, has shown that using wimpy cores and in-order memory systems can provide energy-efficiency with high throughput for some

This work was funded in part by NSF CAREER Award 0846152, NSF Award 1018850, NSF Award 0811794, and NSF Award 1228992.

Requirements	Cloud Systems	Embedded Systems
Energy-efficiency	Limit operating costs	Limit power budgets
High utilization	Limit cost of ownership	Limit area budgets
Resource guarantees	Meet QoS agreements	Real-time tasks

TABLE I. Similarities in Design Requirements for Processors in Cloud and Embedded Systems

of their workloads [3], and similar manycore architectures can provide area and energy efficiency for embedded systems as well. In manycore processors, there are pressures to share the limited on-chip resources in order to simultaneously meet all three demands. Independently providing each processing element sufficient resources to meet its peak demand can provide guarantees, but it would be prohibitively area-expensive. Conversely, dividing up the total resources typically available in manycore processors into fixed partitions may lead to local resource shortages despite sufficient aggregate resources.

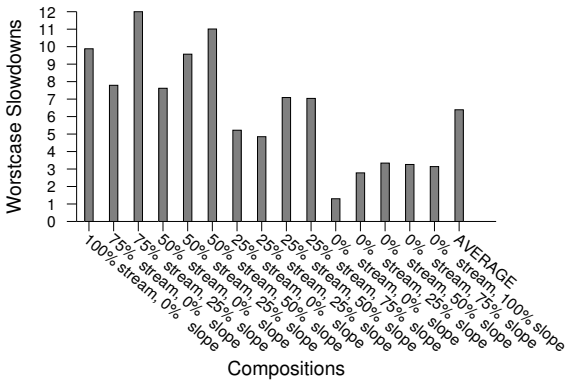

Fig. 1. On a simulated 32-core processor (see Section III for details), we see that slowdowns are both large, 6× on average, as well as highly variable, from less than 2× to as much as 12× in the worst case. Thus, CPU-time should not be used as a proxy for progress when making high-order decisions in embedded systems using space-multiplexed manycore processors.

Sharing these resources allows higher utilization than static partitioning. However, sharing also leads to resource contention, or *interference*. As shown in Figure 1, interference causes uneven and unpredictable slowdowns in application performance, leading to difficulties in monitoring and managing application progress. As a result, decisions made in the presence of interference, such as attempts to maintain performance guarantees, can lead to substantial inaccuracies, as shown by Govindan et al. [4].

Our Approach. In traditional uni-core processors, *CPU-time* provided the notion of interference-free progress, and could be effectively used for tracking performance as well as scheduling or distribution of resources. However, for space-multiplexing

manycore processors, CPU-time does not accurately reflect interference-free progress and is inadequate for these decisions. We propose **Progress Time** as a counterpart of CPU-time for the purpose of tracking application progress in space-multiplexed manycore systems. We define *Progress Time* as **the amount of time required for an application to complete the same amount of work it has done so far, were it to have been allocated *all* CPU resources.**

We describe *TimeCube*, a manycore embedded processor that is augmented by hardware to efficiently support dynamic execution isolation and shadow performance modeling to enable the simultaneous and online estimation of Progress Times for all applications with a high degree of accuracy. TimeCube dynamically partitions last-level cache, memory bandwidth, and DRAM space to enable dynamic execution isolation and uses shadow structures to provide shadow performance modeling and estimate Progress Times efficiently in hardware. TimeCube then uses Progress Times to dynamically reallocate portions of the partitioned critical shared resources (last level cache and memory bandwidth) among applications to increase system throughput while maintaining fairness among applications by maximizing their mean Progress Time. We also dynamically tune the application DRAM prefetchers based on their bandwidth utilization.

Results. Our evaluation shows that the dynamic execution isolation provided in TimeCube enables online estimation of application progress with an average error of less than 1% on a 32-core system, even when the slowdowns witnessed due to interference were 6× on average and as much as 12× in the worst case. The accuracy of Progress Time estimation makes it highly reliable for use in high-order decisions. Our Progress Time-centric resource allocation increases throughput by 36% on average. The results make a compelling case for adding the requisite micro-architectural structures to support Progress Time in manycore chips for embedded systems.

In summary, the paper makes the following contributions:

1) We develop *Progress Time* as a counterpart of application CPU-time for interference-free progress in space-multiplexed systems. Systems can use Progress Time to accurately track online application progress and make high-order decisions.
2) We propose TimeCube, a manycore processor for embedded systems. TimeCube provides online Progress Time estimation with less than 1% error, even in the presence of 6× average slowdown.
3) We propose a Progress Time-based microarchitectural resource allocation scheme that increases throughput by 36% on average when compared to existing allocation schemes.
4) We propose a prefetcher throttling mechanism that tunes the prefetching intensity based on the dynamically allocated bandwidth.

II. TimeCube Design

TimeCube is a manycore processor augmented with hardware mechanisms to efficiently and accurately track application progress. Multiple applications can execute simultaneously on TimeCube; even more than the number of cores, since it supports temporal-multiplexing. Every application executes on an independent core with private L1 data and instruction caches and a DRAM prefetcher. These applications share the last-level caches, memory bandwidth, and DRAM banks, similar to existing commercial manycore processors [5].

The interference resulting from this resource sharing breaks the correspondence between CPU-time and actual performance on space-multiplexed manycore systems, which can lead to erroneous estimations regarding progress of execution, even in the presence of state-of-the-art virtualization techniques [4]. In existing embedded systems, many high-order decisions, such as resource allocation and scheduling, are done in accordance to application progress. Therefore, we propose that these decisions should be made based on application *Progress Times* rather than their CPU-times, where Progress Time is the amount of time required for an application to complete the same amount of work it has done so far, were it to have been allocated *all* CPU resources, as shown in Figure 2.

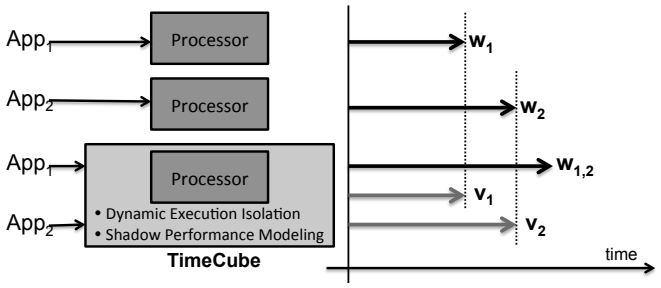

Fig. 2. TimeCube uses dynamic execution isolation and shadow performance modeling to estimate the Progress Times (v_1 and v_2) for applications (App_1 and App_2) running simultaneously for CPU-time $w_{1,2}$. Application Progress Times are equal to their standalone CPU-times (w_1 and w_2).

While offline techniques [6][7] have been proposed to measure the architecture-specific interference between applications, online progress tracking allows us to handle previously unseen applications, handle live input data for known applications, capture phase-specific interference, and provide better online control over application progress rates. As Figure 2 describes, TimeCube provides two key capabilities to enable highly-accurate online estimation of application Progress Time:

- **Dynamic Execution Isolation** ensures that an applications execution, and hence its Progress Time, is not *unpredictably* affected by other concurrently running applications. This can allow us to not only improve Progress Time estimation accuracy but also dynamically control application progress rates. TimeCube includes hardware mechanisms for handling interference for microarchitectural resources, such as cache, memory bandwidth, and memory banks. Conventional runtime software mechanisms, such as virtual machine monitors and hypervisors, can handle interference between I/O threads contending for network bandwidth, or other such system-level resources, by applying thread priorities in the scheduler and/or using backoff algorithms to reduce contention.
- **Shadow Performance Modeling** allows an application's standalone performance to be estimated with a high degree of accuracy using an extrapolation of its isolated execution. TimeCube takes into account

micro-architectural resource usage to accurately estimate Progress Time for a single manycore chip. For taking into account system-level resources, we would need to isolate and estimate performance based on all system components including network and I/O.

Using these two capabilities, TimeCube can eliminate the unpredictable effects of resource interference and estimate standalone application performance with a high degree of accuracy to generate Progress Time.

A. Dynamic Execution Isolation in TimeCube

TimeCube provides dynamic execution isolation by partitioning critical shared resources and dynamically allocating portions of resources to the competing applications after regular intervals, as shown in Figure 3. This partitioning of shared micro-architectural resources eliminates resource interference, and an application's execution is not affected by other concurrently running applications. The allocation is done dynamically to avoid under-utilization of resources, since different applications have different utility for on-chip resources, which can also vary over time. There are many shared resources in manycore architectures, but for this paper we focus on three resources critical to compute workloads: last-level cache, off-chip memory bandwidth, and DRAM space. Contention over memory controllers in an in-order memory system is low, and the situation for our NoC is similar. In a system with high contention for either resource, TimeCube could be extended with fair queuing arbiters [8], or virtual channels, respectively, to provide dynamic isolation over these resources.

Along with the core, each application also gets a dynamically allocated portion of the shared last-level (L2) cache [1], a portion of the shared memory bandwidth and some statically allocated DRAM banks (determined in software). The partitioning and reconfiguration of resources is kept invisible to software, which allows us to use legacy code. The programs execute continuously and uninterrupted even while the resource partitions are being reconfigured, as shown in Figure 3c.

Dynamic Cache Partitioning TimeCube partitions the shared last-level cache between applications to provide dynamic execution isolation. It uses associative cache partitioning, similar to the Virtual Private Caches proposed by Nesbit et al. [8]. TimeCube partitions the cache ways between applications by dynamically assigning a fixed number of associative ways for each application. When an application accesses the cache, all the associative ways are checked. On a cache miss, the data brought in from main memory is placed in the cache. If the number of ways occupied by the requesting application is less than the allocated ways, another application's data is evicted, one that is occupying more ways than allocated. Otherwise, the requesting application's least recently used data is evicted. This maintains cache allocations between applications. However, the applications can still face interference due to the limited cache access bandwidth; therefore, TimeCube multiplexes the cache accesses, as proposed for Virtual Private Caches, based on the fraction of cache-access bandwidth allocated to an application, which is the same as the fraction of cache ways allocated, and maintains dynamic execution isolation.

Dynamic Memory Bandwidth Partitioning TimeCube dynamically partitions the memory bandwidth between applications to reduce interference. Even if an application is given its allocated bandwidth, if the memory scheduling is not done fairly, the applications might have unpredictable slowdowns. We use a fair queueing arbiter [9], which does fair scheduling across applications while staying within their bandwidth quotas. The performance of individual applications can be further improved by using state-of-the-art memory traffic scheduling techniques, which may reorder application memory requests based on prefetcher accuracies [10], or the status of DRAM row buffers [11] etc. In order to limit the possible bandwidth allocations, we *bin* the bandwidth i.e. we allocate bandwidth only in multiples of a fixed percentage of total bandwidth (1%).

Static DRAM Partitioning DRAMs are typically composed of a number of banks that are fronted with a row buffer to reduce access latency on repeated accesses to a line. In order to reduce interference on DRAM banks, TimeCube splits the memory banks statically among the applications along with the corresponding row buffers; however, the number of DRAM banks allocated to an application is not fixed and depends on the amount of memory allocated to the application by the operating system. Thus, an application cannot alter the contents of another application's row buffers to unpredictably affect its memory access time. This bank partitioning is maintained at the memory controllers [2] , and the memory page allocator (OS) allocates pages to applications only on the memory banks assigned to them, as described by Liu et al. [12]. While this is a simpler approach compared to interference-prone performance-preserving techniques like ballooning [13], our experiments showed that bank partitioning does not significantly reduce performance for typical manycore architectures, since the memory-sensitive workloads are bottlenecked at the DRAM pin interface and not the DRAM row buffers.

B. Shadow Performance Modeling in TimeCube

TimeCube calculates Progress Time using an analytical performance model that uses execution statistics collected through shadow hardware structures placed on every core. TimeCube collects the Progress Time for a spectrum of shared resource allocations for all applications, as shown in Figure 4. These collections of Progress Times are called the *Progress Time Tables*, or *pTables*. We believe that pTables provide resource abstraction at the right granularity, i.e. concise enough to be calculated by the microarchitecture during program execution and rich enough to be used in high-order decisions, such as resource management in TimeCube. TimeCube updates the pTables, which are stored in the cores, in parallel with application execution, after regular execution intervals.

We now describe TimeCube's analytical model to calculate Progress Time for application i for an execution interval j, if it were allocated c cache-ways and b bandwidth-bins.

[1]TimeCube is a non-cache coherent architecture like Intel SCC [2]; inter-process coherence is handled by the OS through separate memory allocation.

[2]TimeCube could leave DRAM management to software given support for dynamic execution isolation and shadow performance modeling.

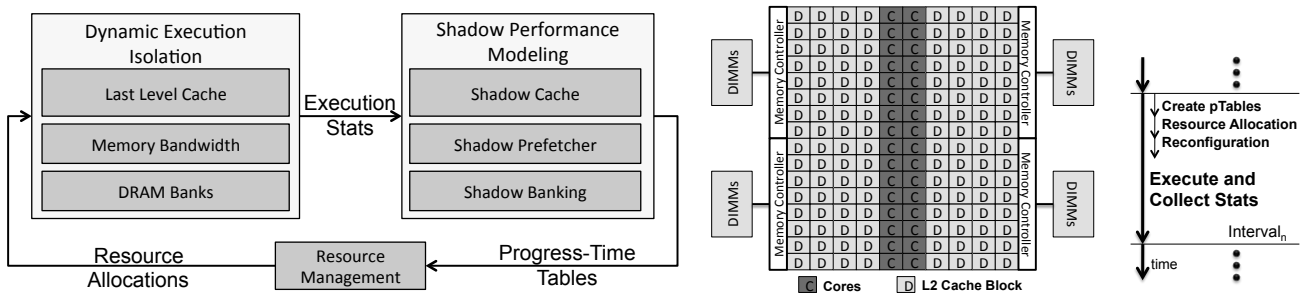

Fig. 3. TimeCube partitions the critical shared resources to provide dynamic execution isolation, and uses execution statistics based analytical performance estimation to provide shadow performance modeling (a). TimeCube is a scalable architecture (b) with spatially distributed cores (C) and L2 cache blocks (D) connected over a network-on-chip, or NoC. TimeCube creates Progress Time Tables, a collection of Progress Times over the entire spectrum of cache and memory bandwidth allocations, which are then used for resource management. Every interval TimeCube collects statistics, creates pTables, and allocates and reconfigures shared resources simultaneously for all applications, all in parallel with execution (c).

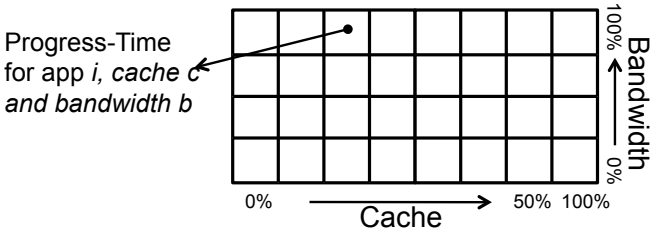

Fig. 4. Progress Time Tables. TimeCube calculates Progress Times for all possible allocations of last-level cache and the memory bandwidth for each application. The bandwidth is *binned* and the cache-ways are allocated in powers of two.

$$
\begin{aligned}
ExecTime_j[i,c] \quad = \quad & const_j + (L2Hit_j[i,c] \times L2HitLatency_j[i,c]) \\
& +(PrefHit_j[i,c] \times PrefHitLatency_j[i,c]) \\
& +(PageHit_j[i,c] \times PageHitLatency_j[i,c]) \\
& +(PageMiss_j[i,c] \times PageMissLatency_j[i,c]) \\
& +(PageCnfl_j[i,c] \times PageCnflLatency_j[i,c])
\end{aligned} \tag{1}
$$

TimeCube's analytical model estimates the hypothetical execution time for the work done by an application in the last interval, but for an arbitrary cache and bandwidth allocation, by estimating the delays caused by the in-order [3] L1 private cache misses in the shared L2 cache, prefetcher and the DRAM (Equation 1).

We assume that the in-core execution time remains unaffected by changing cache and bandwidth allocations. We represent this in-core execution time by $const_j$ for interval j in our analytical model. We collect this value for the current execution by counting the cycles for which the application executed while discarding the cycles spent waiting on L1 private cache misses. We then use this value as the in-core execution time for calculating application Progress Times for the next interval for all possible cache and bandwidth allocations. The time spent inside I/O calls is included within the cycles spent inside the core ($const_j$), not waiting for the memory system. We assume this time to be independent of the cache and memory bandwidth allocation.

To find the time spent in L2 caches we use a shadow cache structure, described below, to estimate the L2 cache hits for the cache size c. We measure the average L2 hit latency for the current cache and bandwidth allocation, and use it for all possible allocations for the next interval. Similarly, we use a shadow prefetching structure, described below, to estimate the number of prefetch hits, while using the average prefetch hit latency from the current execution. When a request misses both L2 and the prefetcher, it is served by the main memory. We measure the DRAM page hit, miss and conflict latencies for the current execution, and use them alongwith the page hit, miss and conflict rates calculated using a shadow DRAM structure, described below, to calculate the remaining components of our analytical model.

With this model we calculate Progress Time for all possible cache and bandwidth allocations for all applications for next interval. According to this model we need to estimate certain shadow L2 cache, prefetcher, and DRAM statistics to estimate the execution times for different resource allocations. We use the following shadow hardware structures to collect these shadow statistics:

- *Shadow-Tags* [15] provide an efficient hardware mechanism to estimate the cache miss rates for any arbitrary cache size. In order to reduce the shadow cache overheads we use set-sampling.
- *Shadow Prefetchers* run a dummy prefetching algorithm by tracking miss streams and launching *fake* prefetches, i.e. while the prefetch request is created, no actual data request is sent to the memory system, and maintain shadow statistics such as prefetches issued, prefetch hit rate, and prefetch hit latency.
- *Shadow Banking* tracks the current state of the DRAM row buffers by modeling DDR behavior for DRAM requests and maintains shadow statistics such as page hits, misses, and conflicts. Our experimental results suggest that shadow banking may not be absolutely essential for this system; using a fixed memory latency imparts on average an error of only 2% in estimating application Progress Times.

We use one shadow-tags structure per core, and one shadow prefetcher and shadow banking structure per cache configuration per core. 40B are required to store an application's pTables. Our experimental results show that these mechanisms

[3]TimeCube has an in-order memory system, like RAW [14], where the core is stalled during its memory miss. Thus, there is no miss concurrency for a single application.

do not have significant area (2.46%) and energy (0.24%) overheads.

The hardware also needs to estimate bandwidth stalls to estimate application performance. We use cache misses and prefetch statistics to calculate the required bandwidth (Equation 2). If the allocated bandwidth exceeds required bandwidth then we assume no bandwidth stalls. Otherwise, the bandwidth stalls are accounted for by reducing performance by the ratio of required and allocated bandwidths (Equation 3). This is based on the assumption that the memory requests are uniformly randomly distributed over program execution.

$$ReqBW_j[i,c] = \frac{L2Misses_j[i,c] + PrefRqs_j[i,c] - PrefHits_j[i,c]}{ExecTime_j[i,c]} \tag{2}$$

$$Performance_j[i,c,b] = \begin{cases} \frac{Instructions_j[i]}{ExecTime_j[i,c]}, if\ ReqBW_j[i,c] \leq b \\ \frac{Instructions_j[i] \times b}{ExecTime_j[c] \times ReqBW_j[i,c]}, otherwise \end{cases} \tag{3}$$

$$pTables_j[i,c,b] = \frac{Performance_j[i,c,b]}{Performance_j[i,c_{total},b_{total}]} \times IntervalTime \tag{4}$$

$$\textit{Progress Time}_i = \sum_{interval\ j} pTables_j[i,c_{alloc},b_{alloc}] \tag{5}$$

Every cell in pTables stores the Progress Time for the corresponding cache and bandwidth allocation by multiplying the interval-time with the ratio of the performance for this allocation and the one with all the cache and memoery bandwidth allocated (c_{total},b_{total}), as shown in Equation 4. TimeCube sums up an application's Progress Times for all past intervals, for the actual cache and bandwidth allocations (c_{alloc},b_{alloc}), to get its total Progress Time (Equation 5).

TimeCube, in-line with existing commercial manycores like Tile64, uses wimpy cores and in-order memory systems to provide energy-efficiency with high throughput. However, performance for out-of-order cores can be modeled as well, as shown by Moreto et al. [16]. Moreover, even though TimeCube is designed for multiprogramming rather than parallel programming, it is reasonable to believe that the techniques outlined here would support consolidated multi-threaded applications as well if given their associated performance models.

Progress Times can also be used on multi-chip embedded systems, and potential processor heterogeneity can be managed in a way similar to existing heterogeneous systems, which calibrate processor performances over a workload. TimeCube can likewise re-normalize the Progress Time estimates over heterogeneous processors.

C. Resource Management in TimeCube

Progress Times can be readily used for high-order decisions in an embedded system; for example, application schedulers can use Progress Times to track application progress and assign higher priorities to applications which are not able to make sufficient progress. The system can allocate resources to applications based on system-level policies, such as high throughput or high fairness, while satisfying SLAs or real-time constraints, such as guarantees of forward progress, minimum execution rate, or maximum slowdown.

At the microarchitectural level, even though manycore processors have large quantities of shared resources such as cache and memory bandwidth, the per core cache size and per core bandwidth is low. Thus, system performance is especially sensitive to memory resource allocation. We can use the pTables to determine a shared resource allocation between the applications based on application characteristics as well as system objectives, such as fairness and high throughput.

In a system with multiple concurrent applications contending for shared resources, an application's progress depends on the amount of resources allocated to it. In a fair system, the progress should be similar between the applications, which means that even if there is a shortage of resources, they are distributed such that the applications which provide lower performance are also given a fair share. However, to attain a high overall system performance, more resources should be given to the applications which provide higher performance. Thus, these two system objectives require conflicting resource distribution strategies.

TimeCube attempts to address the two conflicting goals simultaneously. For every interval j, it finds a cache ($\hat{c}$) and bandwidth ($\hat{b}$) distribution to maximize the geometric mean of application progress, or Progress Times, to find a balance between throughput and fairness.

$$\textit{Mean Progress Time}_{j,\hat{c},\hat{b}} = \prod_i (\textit{Progress Time}_i + pTables_j[i,c,b]) \tag{6}$$

This formulation tries to maximize the forward progress of every application, while reducing unfair slowdowns for applications with lower performance. The arithmetic mean on the other hand would maximize only throughput while absolutely ignoring fairness. After several time intervals, mean Progress Time can be approximated to -

$$\textit{Mean Progress Time}_{j,\hat{c},\hat{b}} = \sum_i \frac{pTables_j[i,c,b]}{\textit{Progress Time}_i} \tag{7}$$

We need an efficient algorithm to maximize the mean Progress Time for the system every time slice. Using the pTables, we can calculate the metric for all possible resource distributions and choose the best allocation; however, this brute force method is inefficient. We employ a dynamic programming based algorithm to calculate the cache and bandwidth allocation that maximizes mean Progress Time. This algorithm is based on the insight that we can reuse the result of a subproblem, i.e. a subset of cache and bandwidth partitioned between a subset of applications to maximize their mean Progress Time. We create a 3 dimensional cube, the *Simultaneous Performance Optimization Table*, or *SPOT*, and use the algorithm, shown in Equation 8, to derive the optimal allocation in the last cell of SPOT, i.e. $SPOTc[N,\$_t,B_t]$. This calculation is done in parallel with execution.

$$\begin{aligned} SPOTv[i,\$,B] &= Max\ possible\ mean\ progress-time \\ &\quad for\ i\ apps,\ \$\ cache,\ and\ B\ bandwidth \\ SPOTc[i,\$,B] &= Cache\ and\ bandwidth\ distribution \\ &\quad for\ SPOTv[i,\$,B] \\ SPOTv[i,\$,B] &= max_{\$',B'}\{SPOTv[i-1,\$-\$',B-B'] \\ &\quad +pTables[i,\$',B']\} \\ SPOTc[i,\$,B] &= SPOTc[i-1,\$-\$',B-B'] \\ &\quad .append([\$',B']_{max}) \\ BestPartition &= SPOTc[N,\$_t,B_t] \end{aligned} \tag{8}$$

In our experiments, we give equal shares of cache and bandwidth to all applications at the start of a run. In a real-world system, applications can be started off with a predetermined fixed starting cache and bandwidth allocation.

For a 32-core TimeCube instance, the hardware allocation mechanism occupies 2.19% of the chip area and 0.23% of the total execution energy. This formulation can handle I/O threads as well, since if a thread is blocked on I/O, the application's pTables will show a low Progress Time for resources, which can then be allocated to other threads. This also elegantly handles more applications than cores because of its additive (rather than multiplicative) formulation. The pTables for the suspended applications are stored within their context.

D. Prefetcher Throttling

When applications are dynamically allocated memory bandwidth, prefetchers need to be dynamically tuned. For example, for a certain cache and bandwidth partition, an application might face a shortage of bandwidth and the bandwidth loss due to incorrect prefetches might overshadow the latency savings because of correct prefetches. We propose a new mechanism which dynamically adjusts the prefetching aggressiveness to maximize the utilization of the dynamically changing available memory bandwidth and can work in conjunction with any existing prefetcher accuracy improvement mechanism.

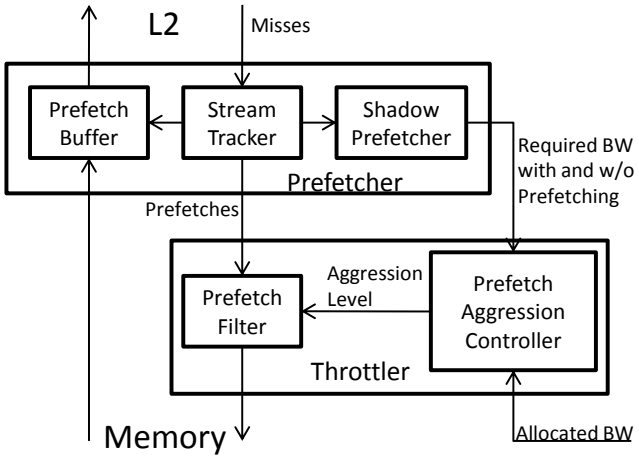

Fig. 5. The prefetcher throttling mechanism changes the prefetcher aggression based on the bandwidth requirements and availability. The prefetch aggression controller (PAC) finds the aggression level to best utilize the available bandwidth, and the prefetch filter drops the corresponding ratio of prefetches.

The prefetch throttler reduces the number of prefetches without affecting the internals of the prefetcher by dropping a fixed ratio of prefetches issued by the prefetcher. This fixed ratio is called the *prefetch aggression level* and it is determined by the *prefetch aggression controller*, or PAC, based on the utility of prefetches to an application, as shown in Figure 5.

The PAC takes in the allocated bandwidth for the application and the required bandwidth for the application with and without prefetching, as calculated by the *shadow prefetchers*. It uses them to determine the required bandwidth at different aggression levels, and assumes that prefetcher accuracy remains the same, leading to a proportional drop in prefetch hits. This provides a rough estimate for the number of memory requests at an aggression level. To determine bandwidth for that aggression level, the PAC determines the overall latency by assuming that the average prefetch hit latency savings remains the same at different aggression levels, and using it to calculate the total latency savings by using the estimated number of prefetch hits. Once the PAC determines the required bandwidth for different aggression levels, it finds the prefetch aggression level for which the required bandwidth is the same or slightly higher than the allocated bandwidth, and the prefetcher accordingly drops a fixed ratio of prefetches during the next execution interval.

Cores	32, x86-64 ISA, 3GHz, superscalar, in-order memory
L1 cache	32KB inclusive, 4 way associative, 8 word line, 1 bank, 3 cycle hit, pipelined, 1 read/write port
L2 cache	128 cache-ways, 1 bank per cache-way, 128KB per bank, 8 word line, 4-way associative, pipelined, 1 read/write port
Network	64-wide, mesh, dynamic router, 1-cycle hop
Prefetcher	stream prefetcher, 128 streams, 32 buffers
Memory	4 controllers, bit-interleaved, 4 DIMMs/channel, 4 Ranks/DIMM, 8 Banks/Rank, 64MB/Bank, 16 Banks and 1GB DDR3 per core, 96Gb/s memory bandwidth

TABLE II. PROCESSOR MODEL

Operation	Energy	Operation	Energy
Instruction Execution	57.2	L2 Data Write	70.9
L1 Tag Match	22.5	Memory Read	5230.1
L1 Data Read	36.0	Memory Write	5120.0
L1 Data Write	38.2	L2 Data Read	65.7
L2 Tag Match	42.2	Network Send	6.2
Progress Time Calc	53.4	Network Rcv	6.4
Shadow-Tag Shift	21.1	Network Hop	4.3

TABLE III. ENERGY (PJ) CONSUMED FOR OPERATIONS

III. TIMECUBE EVALUATION

In this section, we describe our processor model and the benchmarking methodology. We model our evaluation prototype along the lines of commercial manycore processors (e.g. Tile64 [5]). Each core is superscalar, i.e. it can simultaneously execute multiple instructions, but the memory system requests are sent in-order. We use a reconfiguration interval of 25 million cycles. We use PTLsim [17] and a memory-system emulator to simulate execution of multiple applications on a single many-core chip while sharing last level cache and off-chip memory. The emulator internally uses DRAMsim2 [18] for modeling details of the DRAM memory system. Detailed specifications of our evaluation model are presented in Table II. We analytically model the area and power consumption using area and energy numbers obtained from RAW [14] [19] [20] and McPAT [21] scaled to 45nm, as specified in Table III. In order to reduce simulation run times, we extract application representative phases using SimPoint [22] and then concurrently run SimPoint combinations.

Benchmarks and their Classification In order to simulate a typical manycore processor workload, we run combinations drawn from 26 benchmarks that span SPEC2K, SPEC2K6, and an I/O intensive benchmark suite we developed internally to model data-intensive workloads, as shown in Table IV.

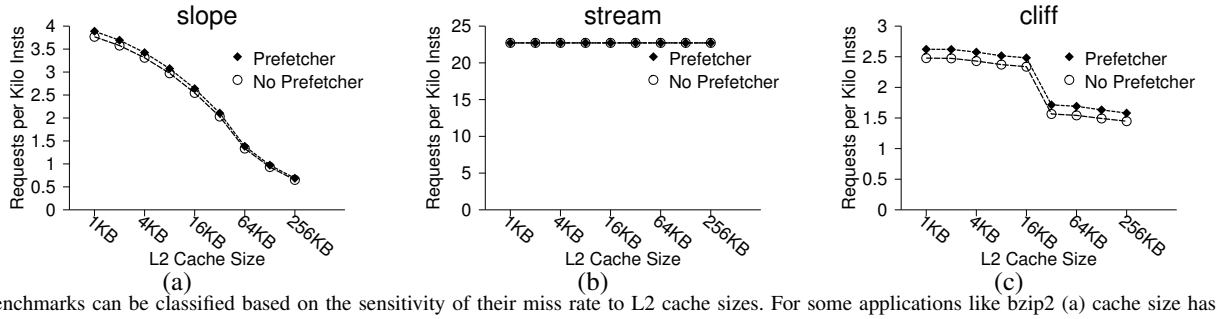

Fig. 6. Benchmarks can be classified based on the sensitivity of their miss rate to L2 cache sizes. For some applications like bzip2 (a) cache size has a steady impact on miss rate, while for others like apsi (b) it has no effect, and some applications like mgrid (c) have a cliff-like profile.

Benchmark	uops /Inst	32KB MPKI	L2Hit 128KB	L2Hit 16MB	Type	Benchmark	uops /Inst	32KB MPKI	L2Hit 128KB	L2Hit 16MB	Type
IO/webCrwlr	1.67	8.01	12.27%	20.52%	slope	IO/faceDetect	1.71	0.27	60.81%	60.96%	strm
IO/fotoBlur	1.69	11.34	10.79%	17.22%	slope	IO/diskBckup	1.70	9.33	12.87%	19.21%	slope
FP2000/wupwise	1.68	15.37	0.07%	1.11%	cliff	FP2000/ammp	1.68	9.14	3.21%	97.25%	slope
FP2000/swim	1.68	28.86	0.00%	56.47%	cliff	FP2000/lucas	1.73	5.38	0.00%	0.04%	strm
FP2000/mgrid	1.68	2.52	0.00%	35.96%	cliff	FP2000/fma3d	1.73	3.44	4.05%	22.53%	cliff
FP2000/applu	1.68	2.56	4.38%	7.87%	strm	INT2000/parser	1.65	8.52	11.32%	97.72%	slope
INT2000/vpr	1.65	11.82	8.99%	87.82%	slope	INT2000/bzip2	1.70	2.21	3.08%	78.48%	slope
FP2000/art	1.68	45.02	0.00%	0.00%	strm	INT2000/twolf	1.65	19.03	3.55%	88.51%	slope
FP2006/equake	1.67	11.41	7.68%	11.89%	strm	FP2000/apsi	1.68	22.64	0.00%	00.00%	strm
INT2006/astar	1.71	1.47	27.36%	40.57%	slope	FP2006/namd	1.71	2.49	62.77%	87.94%	slope
INT2006/bwaves	1.73	0.17	0.39%	2.01%	cliff	INT2006/sjeng	1.70	1.08	53.55%	74.23%	slope
FP2006/h264ref	1.67	1.54	18.57%	59.90%	slope	FP2006/soplex	1.71	2.56	10.19%	56.23%	slope
INT2006/hmmer	1.68	2.60	2.70%	84.69%	cliff	INT2006/specrnd	1.65	0.06	4.17%	4.31%	strm

TABLE IV. BENCHMARK CHARACTERISTICS. WE USE BENCHMARKS THAT PROVIDE A DIVERSE MIX OF MEMORY CHARACTERISTICS SUCH AS MISS RATES IN L1, HIT RATE IN L2, AND CACHE MISS PROFILES.

This selection provides a rich spectrum of cache and memory characteristics, as well as instruction level heterogeneity as shown by uops/inst, and includes applications such as web crawlers, photo filters, face detection, computer aided design tools, scientific computations, data compression, parsing, image recognition, and security algorithms.

The manycore evaluation space, where we run all possible benchmark combinations, is very large. Moreover, it provides no intuition about the benchmarks that we have not included in our evaluation. In order to limit the evaluation space as well as incorporate a structure into our evaluation, we classify our benchmarks according to a three-type taxonomy [4], and then examine runs that include different ratios of the three types. The taxonomy is as follows: An application which sees no drop in miss rate with increasing cache size is a *stream* application, an application which sees a sudden drop in miss rate with cache size is a *cliff* application, and an application whose miss rate drops gradually with increasing cache size is a *slope* application, as described in Figure 6. We can then run representatives of these classes to estimate behavior of similar applications to refine our manycore evaluation space. For our experiments, we run workloads with incrementally changing composition of benchmarks classes. For each composition, we run all possible combinations of benchmarks within every benchmark class, and report the arithmetic mean of their results.

[4]In the applications examined, cache sensitivity was a strong classifier that predicted other characteristics, such as stream applications having good prefetching behavior and high bandwidth requirements. For a workload with high variance within cache sensitivity categories, additional classification axes would be beneficial.

A. TimeCube's pTables are Highly Accurate

In this subsection, we evaluate the mechanisms for creating pTables in TimeCube. For validation of our performance estimation model, we measure an application's estimated standalone performance in concurrent mode and its actual performance in the standalone mode. We ran experiments using the methodology explained previously, and the results show that we are able to estimate an application's standalone performance and its estimated slowdown with 1% average error, as shown in Table V. Thus, we can reliably use TimeCube's pTables for progress measurement and resource management in embedded systems.

strm (%)	**slope** (%)	**cliff** (%)	**error** (%)	**strm** (%)	**slope** (%)	**cliff** (%)	**error** (%)
100	0	0	0.39	25	50	25	0.19
75	0	25	0.41	25	75	0	0.16
75	25	0	0.27	0	0	100	7.04
50	0	50	1.26	0	25	75	1.76
50	25	25	0.51	0	50	50	0.61
50	50	0	0.01	0	75	25	0.40
25	0	75	1.72	0	100	0	0.35
25	25	50	0.49	AVERAGE			1.01

TABLE V. TIMECUBE CREATES PTABLES WITH 1% AVERAGE ERROR OVER A SPECTRUM OF BENCHMARK COMPOSITIONS, AND CAN RELIABLY USE THEM FOR TRACKING PROGRESS AND MANAGING RESOURCES.

B. Increased Resource Utilization

Throughput is a first order concern for concurrent manycore systems. We quantitatively analyze the throughput obtained with Progress Time based resource allocation and compare our scheme against a *baseline* in which we first partition

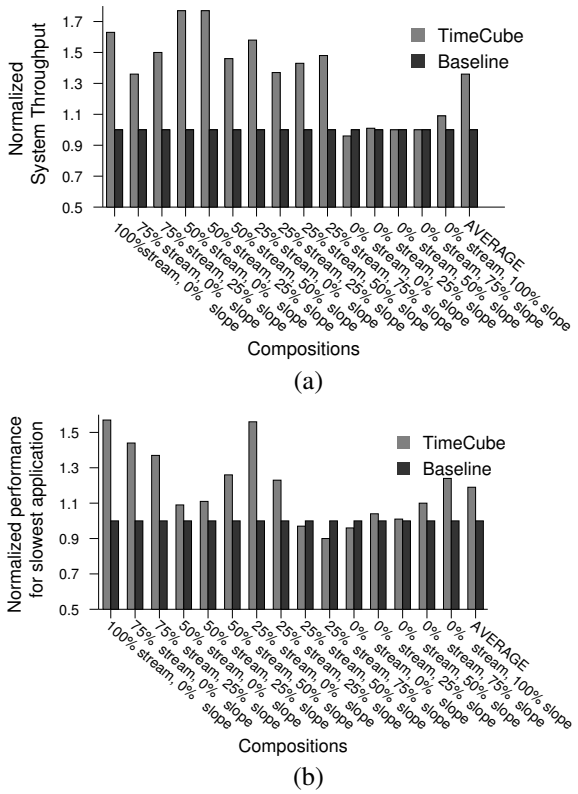

Fig. 7. TimeCube's Progress Time-based resource allocation leads to higher throughput (a) for the system (36% on average), and higher performance (b) for the applications (19% on average) due to better resource utilization.

the caches to minimize the cache misses [15], and then we partition the bandwidth to provide equal slowdown between applications [9]. Our baseline provides a higher performance compared to existing commercial manycore system resource allocations, which provide fair bandwidth sharing between applications, but do not minimize cache miss rates by through demand-based dynamic cache allocation.

We observed that throughput improves by 36% on average for our scheme, compared to the baseline, as shown in Figure 7(a). These gains are made possible due to simultaneously allocating different resources with a shared objective, leading to increased resource utilization, as opposed to existing architectures that end up allocating different resources (cache and bandwidth) with possibly conflicting objectives (throughput and fairness respectively), due to the lack of system-wide online performance metrics, such as the Progress Time. This higher resource utilization also leads to an improvement in application performances by 19% on average, as shown in Figure 7(b). pTables provide the required information that helps us allocate these resources simultaneously and increase utilization.

C. Prefetcher Throttling

We ran experiments over the previously described workload mixes to test the benefits of prefetcher throttling across a range of allocated bandwidths (Figure 10). We use *nine* aggression levels (0-8) in TimeCube. Our experiments show that at lower bandwidths, it is beneficial to turn off prefetching as bandwidth is precious and should not be wasted on potentially bad prefetches. At higher bandwidths, however, we could afford

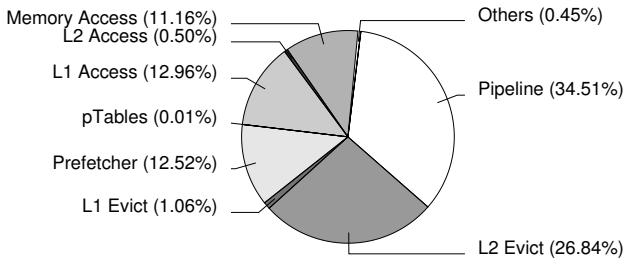

Fig. 8. Energy distribution in TimeCube. Energy consumed by pTables (0.01%) is very small.

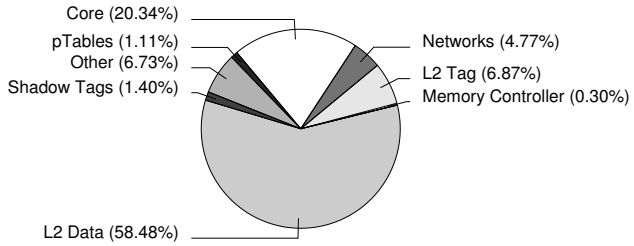

Fig. 9. Area distribution in TimeCube. The area consumed by shadow-tags and pTables is small (1.40% and 1.11%, respectively). Area consumed by resource allocation is 2.19%.

to spend some bandwidth on inaccurate prefetches in lieu of the latency savings of prefetch hits. Our throttling mechanism figures out the *right point* at which to change the prefetcher aggression level. This leads to improved performance over static policies in the regime where the available bandwidth lies in between the bandwidths required with prefetching fully ON or fully OFF. Hence, prefetcher throttling provides a near optimal performance at all bandwidths by approximately tracking the Pareto curve for all throttling levels.

D. Area and Energy Distribution in TimeCube

We now analyze the energy and area distribution for TimeCube. For an example 32 application mix, we observe that the portion of total energy consumed in L2 access is low (0.50%), as shown in Figure 8. Most of the energy is consumed in core execution (47.47% including L1 access) and main memory operations (45.36% for access and writeback). Energy consumed for supporting Progress Time is low, i.e. 0.01%, while the energy consumed in using Progress Time to allocate resources was 0.23%. The microarchitectural mechanisms required to estimate Progress Time consume less than 3% chip area. Shadow-Tags consumes 1.40%, and pTables 1.11%, as shown in Figure 9. Area consumed by hardware resource allocation (2.19%) is small as well. Overall, the mechanisms for measuring and using Progress Time in TimeCube are energy and area efficient.

IV. Related Work

Interference in manycore systems. The emergence of manycore embedded computing, which offers higher density and energy-efficiency, was punctuated by the arrival of Tilera's Tile Gx100 [1] and Intel's 48-core SCC [2]; this has further necessitated the reduction/measurement of interference in embedded systems, as these manycore processors rely on shared resources that greatly impacts application performances, as

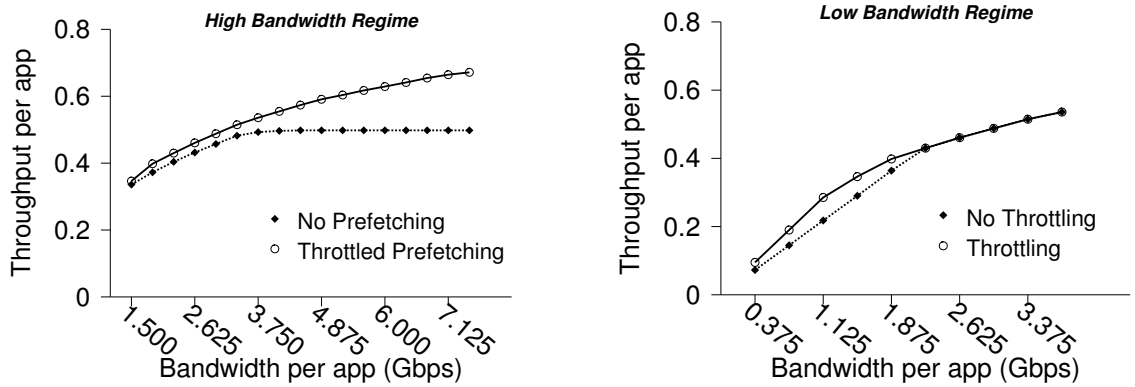

Fig. 10. Prefetcher Throttling maximally utilizes the available bandwidth by intelligently switching between full prefetching, no prefetching, as well as in between aggression levels. When provided with sufficient bandwidth, prefetch throttler sends all requests to memory(a), however for lower bandwidth regimes, prefetch throttler switched off prefetching completely to avoid wasting bandwidth on incorrect prefetches (b). Prefetcher throttling mechanism changes the aggression levels at the *right point* between these two regimes, which leads to a better performance than both no prefetching and full prefetching.

described by Tang et al. [23]. This can lead to difficulties in existing resource management techniques for multiprogrammed embedded systems, such as the ones proposed by Lipari et al. [24], Bernat et al. [25], and Beccari et al. [26]. Govindan et al. [4] show that even with the use of software mechanisms, such as hypervisors, the unpredictability in slowdowns when sharing architectural resources is very high. Stillwell et al. [27] also examined the performance impact of resource sharing in servers at the system level, reducing the effectiveness of techniques such as resource reservation [28] and proportional resource sharing [29] for real-time systems. For resource-sharing embedded systems, it is important to accurately estimate the progress of applications and exercise control over it in order to maintain performance guarantees and improve resource utilization, as also pointed out by Buttazzo [30].

Progress Time provides a performance abstraction for interference-free execution. Such abstractions can be useful for high-order decisions such as resource management and progress tracking, as suggested by Zhang et al. [31], in manycore systems.

Dynamic Execution Isolation. Performance isolation has been proposed as a means to reduce resource interference. Verghese et al. [32] proposed mechanisms for performance isolation for resources such as I/O bandwidth and storage, while Banga et al. [33] suggested resource containers to isolate and account for system-level resource usage. However, since typical manycore architectures rely on shared processor resources, this performance isolation (and not just resource isolation [8]) should be extended to the micro-architectural levels to account for application slowdowns due to sharing of processor resources.

Shadow Performance Modeling. TimeCube creates Progress Times using an analytical performance estimation model similar to the one proposed by Solihin et al. [34]. We further enhance our model by tracking prefetches, adding memory bandwidth constraints by tracking dirty lines, similar to the mechanism proposed by Kaseridis et al. [35], and modeling the details of DRAM DDR protocol and bank buffer behaviors. For our model we need cache miss estimates for arbitrary cache sizes. Shadow cache techniques have been proposed for associative caches, such as by Zhou et al. [36], which are based on the LRU-stacking property [37].

Shared Resource Management. We allocate multiple resources simultaneously in conjunction with resource partitioning schemes. Our allocation scheme is online as opposed to offline profiling based allocation schemes proposed by Liu et al. [6] and Suh et al. [7]. Bitirgen et al. [38] also proposed simultaneous cache and bandwidth allocation using machine learning; however, their technique provides no information about application slowdowns and requires a training phase. Srikantaiah et al. [39] also propose simultaneous resource allocation, but they assume a simple exponentially decaying miss rate with increasing cache size, which is an oversimplification, as shown in Figure 6. Federova et al. [40] examined OS-level scheduling to optimize CMT (multithread CMPs) performance; however, we are able to provide a finer grained control over application execution rates. Previous work has also proposed individual resource allocation; for example, Hsu et al. [41] tune their cache allocation algorithm to maximize different metrics such as fairness and throughput; and Guo et al. [42] allocate cache partitions based on QoS provided by choosing between strict, elastic, and opportunistic schemes.

TimeCube's distributed memory controllers can utilize any memory scheduling scheme which fairly distributes the available bandwidth between applications. There exist many fair scheduling techniques, such as stall-time fairness [43], or self-optimizing controllers [44]. However, we use the fair queue arbiter [8], since it does fair-scheduling while staying within allocated bandwidth limits for each application.

Our proposed dynamic prefetcher throttling mechanism can be used in conjunction with mechanisms that improve prefetcher accuracy or timeliness such as Ebrahimi et al. [45] and Lee et al. [10]. Our motivation for prefetcher throttling is to reduce bandwidth pressure which cannot be significantly reduced by existing mechanisms, such as changing prefetching distance as in FDP [46], which also can be used in conjunction with our mechanism.

V. Conclusion

Manycore processors have to tackle the challenge of interference due to space-multiplexing, which can cause large

and unpredictable slowdowns if left unmanaged. Overcoming this hurdle can improve their usability for embedded systems, which need to accurately measure application progress and maintain guarantees about quality of execution. Progress Time can be used to quantify application progress irrespective of resource heterogeneity. TimeCube, a manycore processor, uses dynamic execution isolation and shadow performance modeling to accurately estimate Progress Time with just 1% error and uses them for resource management, increasing throughput by 36%. Overall, the results argue for adding the requisite micro-architectural structures to support Progress Time in manycore chips for embedded systems.

REFERENCES

[1] R. Schooler, "Tile processors: Many-core for embedded and cloud computing," in *Workshop on High Performance Embedded Computing*, 2010.

[2] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, S. Borkar, V. De, R. C. D. Wijngaart, and T. Mattson, "A 48-Core IA-32 Message-Passing Processor with DVFS in 45 nm CMOS," in *ISSCC*, 2010.

[3] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele, "Many-core key-value store," in *International Green Computing Conference and Workshops*, 2011.

[4] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines," in *SOCC*, 2011.

[5] Bell et al., "TILE64 Processor: A 64-Core SoC with Mesh Interconnect," in *ISSCC*, 2008.

[6] C. Liu et al., "Organizing the last line of defense before hitting the memory wall for CMPs," in *HPCA*, 2004.

[7] G. E. Suh, S. Devadas, and L. Rudolph, "Analytical cache models with applications to cache partitioning." in *HPCA*, 2002.

[8] K. J. Nesbit, J. Laudon, and J. E. Smith, "Virtual private caches," in *ISCA*, 2007.

[9] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *MICRO*, 2006.

[10] C. J. Lee, O. Mutlu, V. Narasiman, and Y. Patt, "Prefetch-aware dram controllers," in *MICRO*, 2008.

[11] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *ISCA*, 2000.

[12] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *PACT*, 2012.

[13] C. A. Waldspurger, "Memory resource management in vmware esx server," *SIGOPS Oper. Syst. Rev.*

[14] M. B. Taylor et al., "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," in *ISCA*, 2004.

[15] M. Qureshi and Y. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *MICRO*, 2006.

[16] M. Moreto, F. Cazorla, A. Ramirez, and M. Valero, "Online prediction of applications cache utility," in *Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS 2007*.

[17] M. T. Yourst, "Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator." in *ISPASS*, 2007.

[18] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "Dramsim: A memory-system simulator," in *SIGARCH Computer Architecture News, September 2005*.

[19] J. S. Kim, M. B. Taylor, J. Miller, and D. Wentzlaff, "Energy characterization of a tiled architecture processor with on-chip networks," in *ISLPED*, 2003.

[20] M. B. Taylor, "The raw processor specification," in *Technical Memo, CSAIL/Laboratory for Computer Science, MIT*, 2004.

[21] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.

[22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS*, 2002.

[23] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *ISCA*, 2011.

[24] G. Lipari and S. K. Baruah, "Efficient scheduling of real-time multi-task applications in dynamic systems," in *Real Time Technology and Applications Symposium*, 2000.

[25] G. Bernat and A. Burns, "Multiple servers and capacity sharing for implementing flexible scheduling," *Real-Time Syst.*, Jan. 2002.

[26] G. Beccari, S. Caselli, and F. Zanichelli, "A technique for adaptive scheduling of soft real-time tasks," *Real-Time Syst.*, Jul. 2005.

[27] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, "Resource allocation using virtual clusters," in *International Symposium on Cluster Computing and the Grid*, 2009.

[28] L. Abeni and G. Buttazzo, "Resource reservation in dynamic real-time systems," *Real-Time Syst.*, Jul. 2004.

[29] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *Real-Time Systems Symposium*, 1996.

[30] G. Buttazzo, "Research trends in real-time computing for embedded systems," *SIGBED Rev.*, 2006.

[31] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen, "Processor hardware counter statistics as a first-class system resource," in *Workshop on Hot Topics in Operating Systems*, 2007.

[32] B. Verghese, A. Gupta, and M. Rosenblum, "Performance isolation: sharing and isolation in shared-memory multiprocessors," in *ASPLOS*, 1998.

[33] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: a new facility for resource management in server systems," in *OSDI*, 1999.

[34] Y. Solihin, V. Lam, and J. Torrellas, "Scal-tool: pinpointing and quantifying scalability bottlenecks in dsm multiprocessors," in *SC*, 1999.

[35] D. Kaseridis, J. Stuecheli, J. Chen, and L. K. John, "A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large cmp systems." in *HPCA*, 2010.

[36] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in *ASPLOS*, 2004.

[37] M. D. Hill, "Aspects of cache memory and instruction buffer performance," Ph.D. dissertation, 1987.

[38] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *MICRO*, 2008.

[39] S. Srikantaiah and M. T. Kandemir, "Srp: Symbiotic resource partitioning of the memory hierarchy in cmps." in *HiPEAC*, 2010.

[40] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, "Performance of multithreaded chip multiprocessors and implications for operating system design," in *Proceedings 2005 USENIX Technical Conference*, 2005.

[41] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, "Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource," in *PACT*, 2006.

[42] F. Guo, Y. Solihin, L. Zhao, and R. Iyer, "A framework for providing quality of service in chip multi-processors," in *MICRO*, 2007.

[43] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO*, 2007.

[44] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *ISCA*, 2008.

[45] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-Aware Shared-Resource Management for Multi-Core Systems," in *ISCA*, 2011.

[46] S. Srinath, O. Mutlu, H. Kim, and Y. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *HPCA*, 2007.