Memory Latency Reduction via Thread Throttling

Hsiang-Yun Cheng*, Chung-Hsiang Lin*, Jian Li†, Chia-Lin Yang*
*Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan
Email: {r96027, f94040, yangc}@csie.ntu.edu.tw
†IBM Austin Research Laboratory, Austin, TX 78750
Email: jianli@us.ibm.com

***Abstract*—Memory Wall is a well-known obstacle to processor performance improvement. The popularity of multi-core architecture will further exaggerate the problem since the memory resource is shared by all cores. Interferences among requests from different cores may prolong the latency of memory accesses thereby degrading the system performance. To tackle the problem, this paper proposes to decouple application threads into compute and memory tasks, and restrict the number of concurrent memory tasks to avoid the interference among memory requests. Yet with this scheduling restriction, a CPU core may unnecessarily stay idle, which incurs adverse impact on the overall performance. Therefore, we develop a memory thread throttling mechanism that tunes the allowable memory threads dynamically under workload variation to improve system performance. The proposed run-time mechanism monitors memory and computation ratios of a program for phase detection. It then decides the memory thread constraint for the next program phase based on an analytical model that can estimate system performance under different constraint values. To prove the concept, we prototype the mechanism in some real-world applications as well as synthetic workloads. We evaluate their performance on real machines. The experimental results demonstrate up to 20% speedup with a pool of synthetic workloads on an Intel i7 (Nehalem) machine and match with the speedup estimated by the proposed analytical model. Furthermore, the intelligent run-time scheduling leads to a geometric mean of 12% performance improvement for real-world applications on the same hardware.**

I. Introduction

The Memory Wall [16], [21] problem has been one of the greatest impediments to system performance improvement. With the ubiquity of multi-core architecture, memory system continues to be a precious system resource that is shared among the cores. As the number of processor cores and chip-level parallelism continue to grow rapidly, the contention on the limited memory resources is expected to become more serious. Therefore, the performance improvement of multi-core systems will be severely limited by the efficiency of memory system.

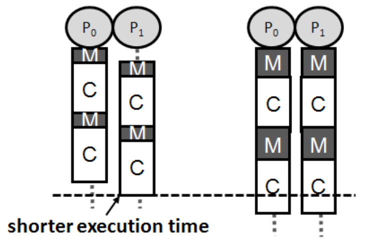
In a conventional interference-oblivious scheduling policy, the contention among concurrent memory accesses from different cores will cause memory access latencies to increase and result in poor system performance [6], [23]. In this paper, we propose to alleviate the contention on the memory system through an interference-aware task scheduling. The main idea of the proposed interference-aware task scheduling is two-fold: First, we decouple applications into equally-sized and cache-friendly compute and memory tasks. In particular, we apply the Stream Programming Model that encourages a *gather-compute-scatter* programming style for convenient memory and computation decoupling [13]–[15]. Second, we schedule these tasks intelligently, based on insights from an analytical model that is corroborated by real-system measurements, to reduce the interference among memory requests thereby improving system performance.

Figure 1(a) illustrates the main idea with two cores scheduling a pool of memory and compute tasks. In this example, only one memory task is allowed to be scheduled at one time. In contrast, a conventional scheme schedules two memory tasks at the same time as shown in Figure 1(b). Due to the interference among memory requests, memory tasks from both cores are slowed down. As a result, the interference-aware scheduling leads to better overall system performance than the conventional scheduling due to the reduction of contention among memory requests. However, a CPU core may unnecessarily stay idle under the scheduling restriction and hurt system performance. Therefore, the challenge lies in making a run-time decision on how many concurrent memory tasks are allowable such that overall system performance is maximized.

To better understand the performance impact of limiting the number of concurrent memory tasks, we derive an analytical model to estimate the performance improvement of the interference-aware scheduling for applications with different characteristics (i.e., memory-to-compute ratios). Given the number of allowable concurrent memory tasks, the analytical model can decide the corresponding speedup compared to an interference-oblivious scheduling policy. This analytical model helps us to decide the memory task scheduling constraint that leads to improved performance. The proposed analytical model is corroborated with a set of synthetic benchmarks with a wide range of memory-to-compute ratios on an Intel i7 (Nahelam) machine. The experimental results indicate that the speedup estimated by the analytical model matches well with that measured on real machines. The results also show that restricting the number of simultaneous memory tasks provides up to 20% performance improvement.

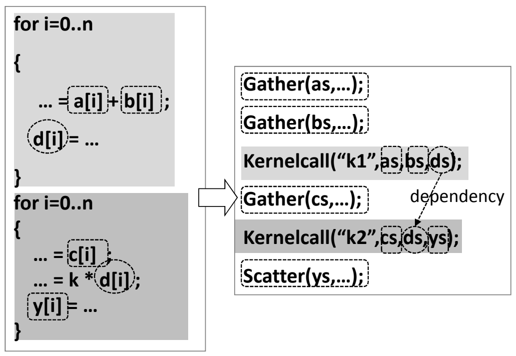
Based on the analytical model, we develop a memory thread throttling mechanism that tunes the allowable memory threads dynamically to adapt to the workload variation. The proposed run-time mechanism monitors the memory-to-compute ratios of a program for phase detection, and then decides the memory

This work was performed in part while Hsiang-Yun Cheng was an intern at IBM Austin Research Laboratory under the sponsorship of ROC National Science Council NSC 98-2911-I-002-077. Hsiang-Yun Cheng is now with the Computer Science and Engineering Department, The Pennsylvania State University.



(a) interference-aware scheduling (b) conventional scheduling

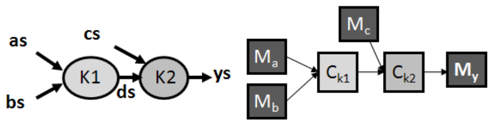
Figure 1: Interference-aware vs. conventional scheduling.



```
for i=0..n
{
  ... = a[i] + b[i] ;
  d[i] = ...
}
for i=0..n
{
  ... = c[i] ;
  ... = k * d[i];
  y[i] = ...
}
```

```
Gather(as,...);
Gather(bs,...);
Kernelcall("k1",as,bs,ds);
Gather(cs,...);
Kernelcall("k2",cs,ds,ys);
Scatter(ys,...);
```

(a) Original code vs. stream programming model



(b) Data flow graph (c) Task graph

Figure 2: Stream programming example.

thread constraint for the next program phase with the help from the analytical model. For each phase in the application, we utilize the analytical model to estimate and compare performance improvements under candidate memory thread constraints, and choose the one that provides the highest performance speedup. The experimental results show that the run-time memory thread throttling mechanism provides about 12% speedup for realistic applications on an Intel i7 (Nahelam) machine.

The rest of the paper is organized as follows. Section II introduces the basic concept of the stream programming model. Section III presents the main idea of the interference-aware task scheduling and the motivation of this work. Section IV describes the proposed run-time memory thread throttling mechanism. Section V and Section VI show our experimental methodology and experimental results. Section VII summarizes related works. Finally, Section VIII concludes the paper.

II. Background on Stream Programming Model

Stream programming advocates a programming style that decouples computation and memory accesses to express the parallelism inherent in a program, and tries to keep the intermediate results locally during a local computation to reduce off-chip memory accesses [13]–[15]. We develop our interference-aware scheduling based on the stream programming model, however our mechanism is not limited to specific stream programming languages. Figure 2(a) shows an example pseudo code that illustrates the *gather-compute-scatter* style in stream programming. First, data are *gathered* into data streams from arrays in memory. For instance, elements of array "a" from index 0 to n are gathered into data stream "as". Then, one or more kernels *compute* the gathered data and store all intermediate results locally. For example, the computation in the original "for" loops are transformed into kernel "k1" and "k2" in the stream code, and the result, stream "ys", is stored locally. Finally, the results, such as elements in stream "ys" are *scattered* back to array "y" in the memory.

Many media and general-purpose applications [7], [13]–[15], [26], like SpecFP2006 and Berkeley dwarfs [15], that are suitable to multi-core chips can be written in the streaming style. Several studies have successfully re-written applications in such a programming style, and demonstrated its efficiency on multi-core architectures [26] [25]. For example, some media applications, such as jpeg/mpeg decoder, and image processing kernels are re-written in the StreamIt language [25]. These applications usually have large data sets and producer-consumer locality. We believe that the *gather-compute-scatter* programming style is applicable to a wide range of applications on multi-cores.

In this paper, *memory tasks* refer to the *gather* and *scatter* operations. *Compute tasks* refer to *compute* operations. Figure 2(b) illustrates the data flow graph of the codes in Figure 2(a). Figure 2(c) shows its corresponding memory/compute task graph. Each *memory task* fetches data from off-chip memory to on-chip memory, which corresponds to on-chip caches on general-purpose processors. Therefore, the corresponding *compute task* can access data directly from on-chip caches without cache misses. Ideally, the data brought in by *memory tasks* should fit into the last-level cache, such that a compute task should see very few cache misses. That is, *memory tasks* dominate off-chip data accesses.

III. Motivation

In the context of the *gather-compute-scatter* style of the stream programming model, we propose to alleviate the multi-core memory bottleneck through an interference-aware task scheduling method. The main idea of the proposed task scheduling is to avoid extensive interferences among memory requests and improve multi-core performance through restricting the number of concurrent memory tasks. More precisely, we call the *Memory Task Limit*(*MTL*), the number of memory tasks that can be scheduled simultaneously. In this terminology, *MTL=1* means that only one memory task is scheduled at a time, and so on. Note that, while the memory task of an application thread may be throttled, the

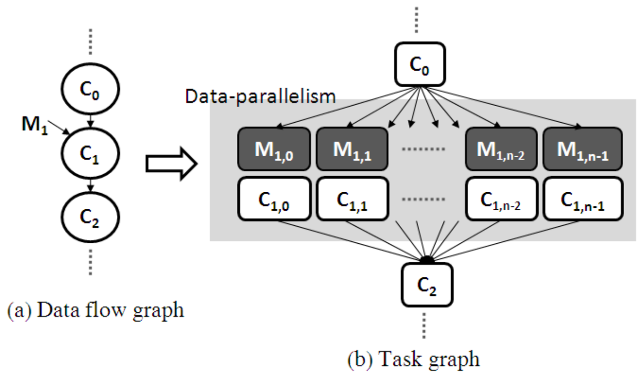


Figure 3: Task partition of a streaming program.

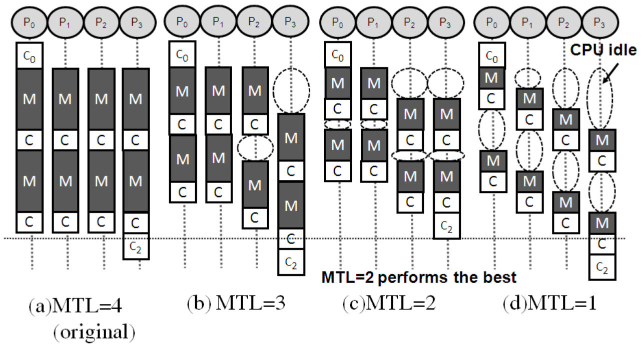


Figure 4: Example workload that performs the best under $MTL = 2$.

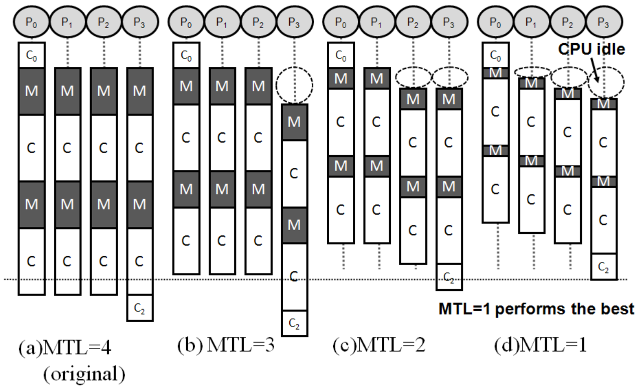


Figure 5: Example workload that performs the best under $MTL = 1$.

application thread itself does not have to stall if it has compute work to do.

We use an example to illustrate the idea of the proposed scheduling. Figure 3 shows a snippet of the task graph of a threaded application that exploits data-level parallelism. Following the stream programming guideline, task C_1 and its companion M_1 in Figure 3(a) are parallelized by forking n equally-sized memory tasks ($M_{1,0}$ to $M_{1,n-1}$) and their n dependent compute tasks ($C_{1,0}$ to $C_{1,n-1}$), which are also equally-sized, as shown in Figure 3(b).

Figure 4 illustrates the scheduling of the task graph in Figure 3(b) on a quad-core processor with *MTL* varying from 4 to 1.[1]. Figure 4(a) shows the conventional scheduling where memory tasks are not throttled, i.e., *MTL* equals to 4, which is the same as the number of cores on the processor. When *MTL* equals to 3 (Figure 4(b)), processor P_3 stalls its memory task due to the scheduling constraint, until one of the other processors finishes its memory task. In this case, memory contention is less compared to the case when *MTL* equals to 4. As a result, the execution time of each memory task is shorter than that in Figure 4(a). Although the reduction in the execution time of memory tasks brings positive effect on overall performance, the downside is that a core may be forced to stay idle due to the *MTL* constraint(i.e., idle time marked by circles in Figure 4). In this example, we can observe that $MTL = 1$ performs worse than $MTL = 4$ due to too much CPU idle time, while $MTL = 2$ performs better than $MTL = 4$.

Due to the tradeoff between the reduction in memory contention and the adverse effect incurred by the idle cores, the *MTL* value that delivers the best performance varies for applications with different characteristics. Figure 5 shows the scheduling of an application with a smaller memory-to-compute ratio than that of Figure 4. In Figure 5, the application has more computations than that in Figure 4, such that there are enough compute tasks to keep all cores busy with a smaller *MTL* value. We can see that the highest performance occurs when $MTL = 1$ for this application.

From the discussion, we know that selecting the best MTL value is challenging due to the complicated interplay between CPU idle time and memory-to-compute ratios of applications. In this paper, we propose a run-time memory thread throttling mechanism that monitors an application's behavior dynamically, and selects the appropriate MTL value in different program phases. The next section details the proposed memory thread throttling mechanism.

IV. Run-time Memory Thread Throttling

The objective of the memory thread throttling mechanism is to dynamically tune the *MTL* for the best performance with different workload behaviors. To better understand the performance impact of different *MTL*s, we first simplify the scheduling behavior through some reasonable assumptions to derive an analytical model for performance estimation. Given *MTL*, say $MTL = k$, and the average execution time of memory task (T_{m_k}) and compute task (T_c), the analytical model can decide whether all cores are busy

[1]Note that the scales of memory task execution times for different *MTL* values are not exact in this example.

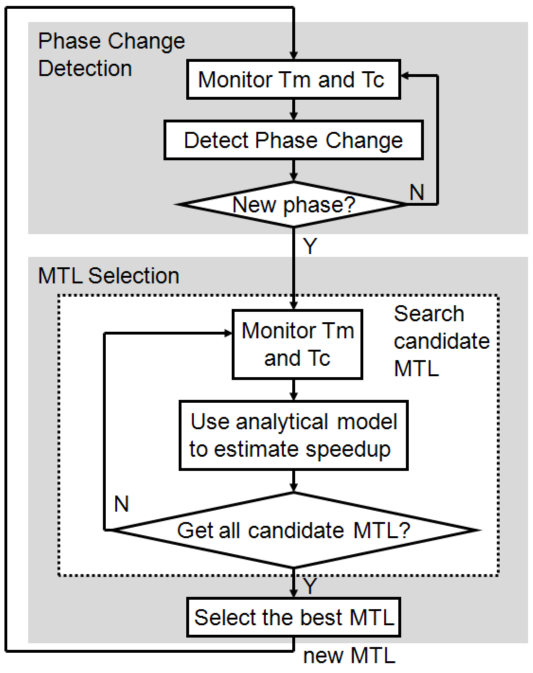


Figure 6: Overview of run-time memory thread throttling.

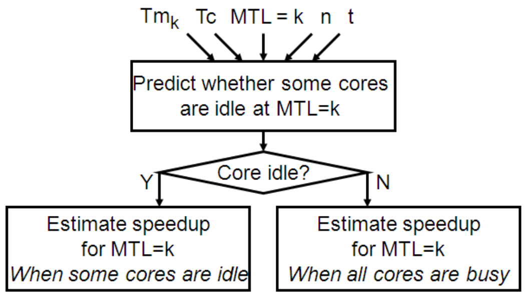

Tm_k Tc MTL = k n t
Predict whether some cores are idle at MTL=k
Core idle?
Y
N
Estimate speedup for MTL=k When some cores are idle
Estimate speedup for MTL=k When all cores are busy

Figure 7: Overview of analytical model.

under $MTL = k$, and the corresponding speedup compared to an interference-oblivious scheduling policy. Based on the analytical model, the proposed run-time mechanism monitors the memory-to-compute ratio of a program for phase detection, and then decides the target *MTL* for performance improvement in the next program phase.

Figure 6 shows an overview of the proposed memory thread throttling mechanism. We use the change in the memory-to-compute ratio as the criterion for phase change detection. We first monitor the execution time of a few memory-compute task pairs and get the average execution time among these tasks. We then apply the proposed *Phase Change Detection* algorithm repeatedly to detect phase change. If a phase change occurs during the execution of the application, the *MTL Selection* algorithm

Table I: Definitions of Symbols

MTL	number of memory tasks that can be scheduled simultaneously
T_{m_k}	average execution time of memory tasks under $MTL = k$
T_c	average execution time of compute tasks
n	number of processor cores
k	selected *MTL*
t	number of memory-compute task pairs
W	number of memory-compute task pairs that are monitored for T_{m_k} and T_c estimation
T_{ml}	contention-free memory latency
T_{ql}	queuing latency when MTL = 1
MTL_{NoIdle}	minimum MTL at which all cores are busy
MTL_{Idle}	maximum MTL at which some cores are idle

utilizes the analytical model to estimate the maximal performance improvement and decide the target *MTL* constraint. The estimation of speedup based on the analytical model also requires monitoring statistics from a few task pairs at different candidate *MTL*s. After selecting the target *MTL*, we continuously monitor and detect phase change in the application to guarantee prompt adaptation of *MTL* at different phases. In the following sections, we describe the analytical model, the phase change detection policy, and the *MTL* selection algorithm in details.

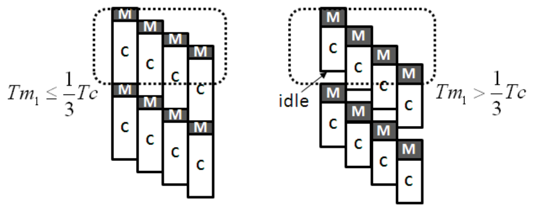
A. Analytical Model

In this subsection, we propose an analytical model to analyze and estimate performance speedup under different *MTL* constraints in the steady state of the proposed task scheduling. First, we sketch the overview of the model. Then, we study the boundary condition that divides the performance formulation into two cases. Finally, we develop the performance model for both cases. The definitions of symbols used in the analytical model are summarized in Table I.

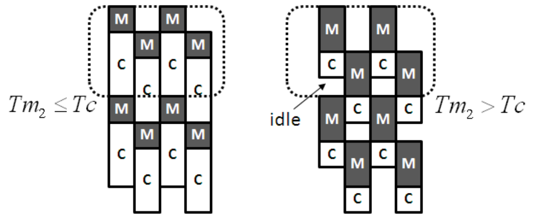
Figure 7 shows an overview of the analytical model. We define T_{m_k} as the measured average execution time of memory tasks (the first subscript m) under $MTL = k$ (the second subscript k). T_c is the measured average execution time of compute tasks. Since compute tasks have little off-chip accesses after all their data are brought to caches by the corresponding memory tasks, their execution time is invariant to *MTL* selection. Since all memory and compute tasks are equally sized in our implementation, we expect all T_{m_k} and T_c are equal among the application threads. We use n to denote the number of processor cores, and t to represent the number of memory-compute task pairs in an application. To estimate the performance under the MTL constraint k, the proposed analytical model first decides whether $MTL = k$ causes cores to idle or not using the following simple inequality:

$$\begin{cases} \frac{T_{m_k}\times t}{k} > \frac{(T_{m_k}+T_c)\times t}{n} \Rightarrow \frac{T_{m_k}}{T_c} > \frac{k}{n-k} & \text{some cores are idle} \\ \frac{T_{m_k}\times t}{k} \le \frac{(T_{m_k}+T_c)\times t}{n} \Rightarrow \frac{T_{m_k}}{T_c} \le \frac{k}{n-k} & \text{all cores are busy} \end{cases} \tag{1}$$

The above inequality compares the time required to finish all the memory tasks under the scheduling restriction of $MTL = k$ (i.e.,$\frac{T_{m_k}\times t}{k}$) and the execution time of the ideal scheduling when



(a) Example: n=4, MTL=1



(b) Example: n=4, MTL=2

Figure 8: Illustration of core idle behaviors.

memory tasks and compute tasks are scheduled one after another (i.e., $\frac{(T_{m_k}+T_c)\times t}{n}$). If the former is larger than the latter, some cores are idle before all the memory tasks are finished, and vice versa. Figure 8 shows two examples ($MTL = 1$ and $MTL = 2$) in a quad-core processor. When $MTL = 1$, all cores are busy if $T_{m_1} \le \frac{1}{3}T_c$. When $MTL = 2$, all cores are busy if $T_{m_2} \le T_c$.

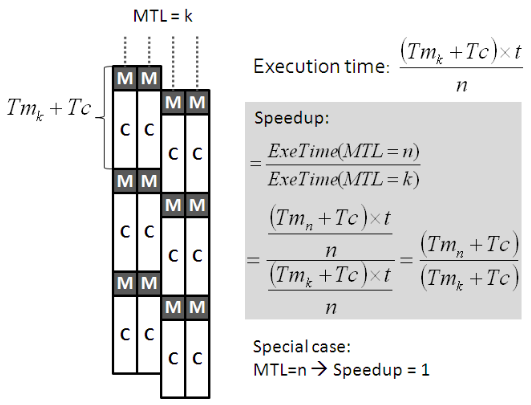
When all cores are busy at $MTL = k$, memory tasks and the corresponding compute tasks are scheduled one after another without gaps, as shown in Figure 9(a). The execution time in this case is $\frac{(T_{m_k}+T_c)\times t}{n}$. The performance speedup over the case of $MTL = n$ without throttling, equals to $\frac{T_{m_n}+T_c}{T_{m_k}+T_c}$.

On the other hand, when some cores are idle at $MTL = k$, the execution time is $\frac{T_{m_k}\times t}{k}$, as shown in Figure 9(b). Correspondingly, the performance speedup equals to $\frac{(T_{m_n}+T_c)\times k}{T_{m_k}\times n}$. Note that, such speedup is not obvious from the time measurement of individual memory and compute tasks.

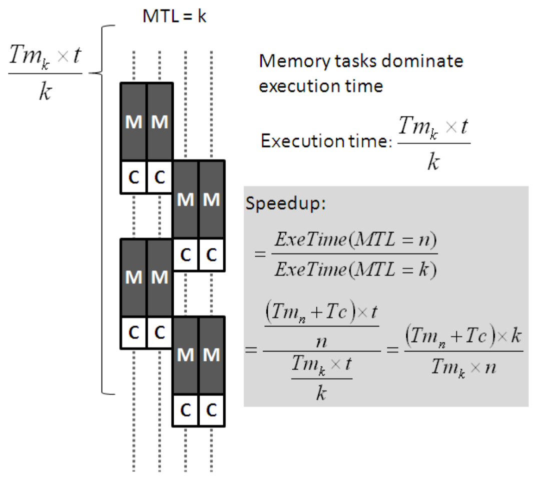
From the analytical model described above, we can estimate performance speedup of the proposed task scheduling under different *MTL* constraints. This analysis gives us an insight on deciding the *MTL* value that provides the highest performance improvement for applications with different characteristics. In the following section, we utilize the analytical model to dynamically select *MTL* for the best system performance.

B. Phase Change Detection

We use the change in the memory-to-compute ratio as the criterion for phase change detection. A naive solution is triggerring *MTL* selection as long as the memory-to-compute ratio changes. However, not each distinctive memory-to-compute ratio maps to different target *MTL*s. This naive solution may lead to unnecessary triggering of *MTL* selection and hurt overall performance. Therefore, we propose a coarse-grained phase change detection policy


MTL = k
$Tm_k + Tc$
Execution time: $\frac{(Tm_k + Tc)\times t}{n}$
Speedup:
$= \frac{ExeTime(MTL=n)}{ExeTime(MTL=k)}$
$= \frac{\frac{(Tm_n+Tc)\times t}{n}}{\frac{(Tm_k+Tc)\times t}{n}} = \frac{(Tm_n+Tc)}{(Tm_k+Tc)}$
Special case:
MTL=n → Speedup = 1
M
C

(a) All cores are busy at MTL = k


MTL = k
$\frac{Tm_k \times t}{k}$
Memory tasks dominate execution time
Execution time: $\frac{Tm_k \times t}{k}$
Speedup:
$= \frac{ExeTime(MTL=n)}{ExeTime(MTL=k)}$
$= \frac{\frac{(Tm_n+Tc)\times t}{n}}{\frac{Tm_k\times t}{k}} = \frac{(Tm_n+Tc)\times k}{Tm_k\times n}$
M
C

(b) Some cores are idle at MTL = k

Figure 9: Performance model at MTL=k

to avoid unnecessary triggering of *MTL* selection.

This phase change detection policy triggers new *MTL* selection only if the change in the memory-to-compute ratio leads to different core idle behaviors. We use a quad-core system to illustrate the basic idea of the policy. Suppose current T_{m_1}/T_c is equal to 0.1, which indicates that all cores are busy at *MTL=1* according to the analytical model. If T_{m_1}/T_c changes from 0.1 to 0.5, the scheduling constraint will cause some cores to idle at *MTL=1*. The policy will then detect it as phase change and trigger new *MTL* selection. Figure 10 shows the overview of the phase change detection policy. We assume that current task scheduling is under the scheduling constraint of $MTL = k$. First, the execution time of W memory-compute task pairs are monitored to estimate T_{m_k} and T_c. Given T_{m_k} and T_c, we utilize the analytical model to estimate the lowest value of *MTL* at which all cores are busy. We define this threshold value of *MTL* as *IdleBound* (i.e., all cores are busy at $MTL \ge$

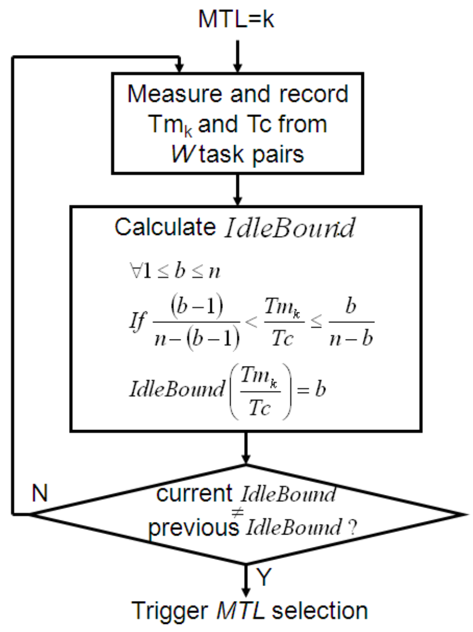

MTL=k
Measure and record Tm_k and Tc from *W* task pairs
Calculate $IdleBound$
$\forall 1 \le b \le n$
$If \frac{(b-1)}{n-(b-1)} < \frac{Tm_k}{Tc} \le \frac{b}{n-b}$
$IdleBound\left(\frac{Tm_k}{Tc}\right) = b$
current $IdleBound \neq$ previous $IdleBound$?
N
Y
Trigger *MTL* selection

Figure 10: Overview of phase change detection.

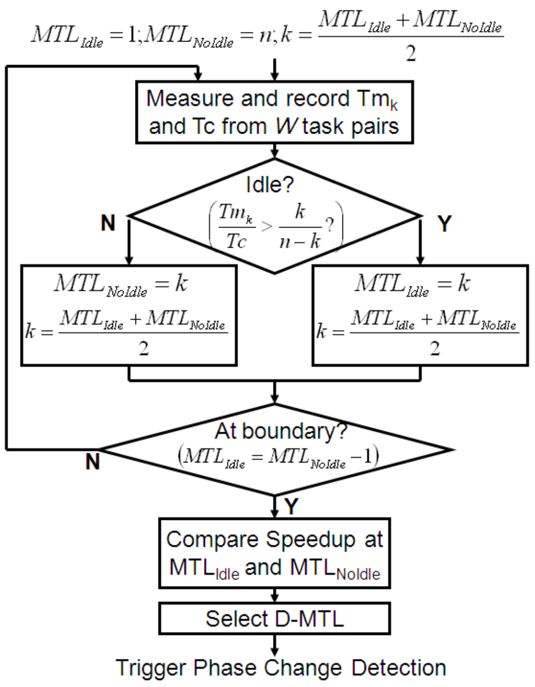

$MTL_{Idle}=1; MTL_{NoIdle}=n; k=\frac{MTL_{Idle}+MTL_{NoIdle}}{2}$
Measure and record Tm_k and Tc from *W* task pairs
Idle? $\left(\frac{Tm_k}{Tc} > \frac{k}{n-k} ?\right)$
N
Y
$MTL_{NoIdle}=k$ $k=\frac{MTL_{Idle}+MTL_{NoIdle}}{2}$
$MTL_{Idle}=k$ $k=\frac{MTL_{Idle}+MTL_{NoIdle}}{2}$
At boundary? $(MTL_{Idle}=MTL_{NoIdle}-1)$
N
Y
Compare Speedup at MTL_{Idle} and MTL_{NoIdle}
Select D-MTL
Trigger Phase Change Detection

Figure 11: Flow chart of *MTL* selection algorithm.

$IdleBound$).

If the core idle behavior (*IdleBound*) is different from the behavior in previous W task pairs, the algorithm will trigger new *MTL* selection. The coarse granularity of phase change detection provided by this algorithm leads to little overheads in *MTL* selection.

C. MTL Selection

For each phase with the distinctive memory-to-compute ratio, we can utilize the analytical model described in Section IV-A to estimate the performance speedup at different *MTL* constraints and select the one that leads to the best system performance. The most naive solution is estimating performance speedup from $MTL = 1$ to $MTL = n$ through the analytical model and selecting the one with the highest performance improvement. However, estimating performance speedup from $MTL = 1$ to $MTL = n$ requires large amount of T_{m_k} and T_c monitoring overheads. That is, we cannot decide the *MTL* constraint for the best performance until monitoring $n \times W$ task pairs for T_{m_k} and T_c. Therefore, we prune the solution space of candidate *MTL*s based on the analytical model to reduce the monitoring overheads of T_{m_k} and T_c. In this subsection, we propose a *MTL* selection method that compares only performance speedup between "the lowest *MTL* at which all cores are busy" and "the highest *MTL* at which some cores are idle" through the analytical model and chooses the *MTL* constraint that leads to higher performance improvement among them.

For those *MTL* constraints at which all cores are busy, the lowest *MTL* should give the best performance according to the analytical model. The reason is as follows. Assume all cores are busy at both $MTL = a$ and $MTL = a + 1$, the performance speedup at $MTL = a$ is equal to $\frac{T_{m_n}+T_c}{T_{m_a}+T_c}$ as explained in Section IV-A. Similarly, the performance speedup at $MTL = a + 1$ is equal to $\frac{T_{m_n}+T_c}{T_{m_{a+1}}+T_c}$. Therefore, $\frac{Speedup(MTL=a+1)}{Speedup(MTL=a)} = \frac{T_{m_a}+T_c}{T_{m_{a+1}}+T_c}$. T_c at different *MTL* constraints are equal due to few last-level cache misses of the equally-sized compute tasks. Since the interference among memory requests at $MTL = a$ is less than or equal to the interference at $MTL = a + 1$, T_{m_a} is less than or equal to $T_{m_{a+1}}$. Hence, $Speedup(MTL = a + 1)$ is less than or equal to $Speedup(MTL = a)$. By induction, one can show that $MTL = a$ performs the best among all $MTL \ge a$ where all cores are busy.

On the other hand, for those *MTL*s at which some cores are idle, the highest *MTL* among them performs the best according to the analytical model. [2] The reason is as follows. Assume some cores are idle at both $MTL = b$ and $MTL = b + 1$, the performance speedup at $MTL = b$ is equal to $\frac{(T_{m_n}+T_c)\times b}{T_{m_b}\times n}$. Similarly, the performance speedup at $MTL = b+1$ is equal to $\frac{(T_{m_n}+T_c)\times(b+1)}{T_{m_{b+1}}\times n}$. Therefore, $\frac{Speedup(MTL=b+1)}{Speedup(MTL=b)} = \frac{T_{m_b}}{T_{m_{b+1}}} \times \frac{(b+1)}{b}$. T_{m_b} includes two components: (1) a contention-free memory latency that is constant, T_{ml}; and (2) approximated queuing latency, $b \times T_{ql}$. We assume the queuing latency at *MTL=1* is a constant represented by T_{ql}, and the

[2]This statement is confirmed by experiments with the studied workloads on quad-core machines.

```
MemoryTasks:
        for(i=start; i<end;i++){A[i]=Const;}
ComputeTasks:
        for(k=0; k<count;k++)
        for(i=start;i<end;i++){A[i]+=k;}
```

Figure 12: The kernel code of the synthetic workload.

total queuing latency is proportional to the number of concurrent memory tasks. Therefore, we have $\frac{T_{m_b}}{T_{m_{b+1}}} = \frac{T_{ml}+b\times T_{ql}}{T_{ml}+(b+1)\times T_{ql}} > \frac{b}{b+1}$. As a result, $\frac{Speedup(MTL=b+1)}{Speedup(MTL=b)} > 1$. By induction, one can show that $MTL = b$ performs the best for all $MTL \leq b$ where some cores are idle.

Based on the above discussion, we propose a dynamic *MTL* selection policy that compares performance at only two *MTL*s to select the *MTL* with the best performance. We use *D-MTL* to denote the dynamically selected *MTL*. Figure 11 illustrates the overview of the algorithm. First, we perform binary search and monitor T_{m_k} and T_c to find "the minimum *MTL* at which all cores are busy" (i.e., MTL_{NoIdle} in Figure 11) and "the maximum *MTL* at which some cores are idle" (i.e., MTL_{Idle} in Figure 11). Then, we compare speedup at MTL_{NoIdle} and MTL_{Idle} through the analytical model to get the *D-MTL* with the best performance. After deciding *D-MTL*, the algorithm will trigger phase change detection mechanism to see whether a new *D-MTL* selection is required.

Pruning the solution space from n different *MTL* constraints to only two *MTL* constraints reduces the monitoring overhead of T_{m_k} and T_c, thus provides a better solution than exhaustively searching all possible *MTL* constraints. Experiments in Section VI show that the proposed policy provides an efficient method to select *D-MTL* for each phase in the applications.

V. Experimental Setup

We implement our run-time memory thread throttling mechanism at the application layer and evaluate its performance on an Intel® Nehalem multi-core machine (Dell® Vostro 430 with processor i7-860 and two DDR3 memory channels). We first create a synthetic micro-benchmark of simple array computation with the stream programming model. We use it to correlate the analytical model with experimental results and observe the potential benefit of the interference-aware scheduling. The kernel code of the synthetic workload is shown in Figure 12. Each memory task executes store operations to initialize the array elements, while each compute task adds a constant to each array element which was brought into the last-level cache by the corresponding memory task. To observe the speedup with different memory-to-compute ratios, we adjust the variable "count" in the compute tasks to create synthetic workloads with a large range of T_{m_1}/T_c (0.01 to 4.00, increment by 0.01). We also vary the memory footprint of each memory task to observe the impact from the capacity misses on the scheduling.

In addition to the synthetic micro-benchmarks, we also consult the prior work by Gummaraju et al. [15] to rewrite three real-world workloads into the stream programming style. They are the

Table II: Workload characteristics: memory-to-compute Ratio (T_{m_1}/T_c)

Benchmark	Name	T_{m_1}/T_c
dft in OpenCV	dft	12.77%
streamcluster in PARSEC	SC_ d128 (*native*)	37.14%
	SC_ d72	43.09%
	SC_ d48	28.90%
	SC_ d36	54.13%
	SC_ d32	24.59%
	SC_ d20	49.58%

Table III: Memory-to-compute ratio (T_{m_1}/T_c in parallel functions in $SIFT$)

Function	T_{m_1}/T_c
COPYUP	21.02%
ECONVOLVE	70.04%
ECONVOLVE2	7.83%
ECONVOLVE3-0	8.45%
ECONVOLVE3-1	8.45%
ECONVOLVE3-2	8.32%
ECONVOLVE3-3	8.27%
ECONVOLVE3-4	8.15%
ECONVOLVE4-0	11.87%
ECONVOLVE4-1	11.66%
ECONVOLVE4-2	12.10%
ECONVOLVE4-3	11.68%
ECONVOLVE4-4	11.53%
DOG	60.32%

dft kernel function from OpenCV [1], streamcluster from PARSEC benchmark suite [4] and Scale-Invariant Feature Transform (SIFT) [2], [20]. To analyze the performance of applications with different memory-to-compute ratios due to different input sets, we vary the size of the input array of streamcluster to create six different streamcluster instances (array dimension sizes:128, 72, 48, 36, 32, 20). [3] Table II shows the T_{m_1}/T_c for dft and streamcluster. Since SIFT is composed of different parallel functions, we report T_{m_1}/T_c for each function in Table III.

We use Intel's prefetch intrinsic (void _mm_prefetch (char const* a, int sel)) to bring data to the last-level cache in memory tasks. The memory footprint of each memory task is always less than the last-level cache size per core to eliminate the influence of capacity and conflict misses in compute tasks. At the beginning of each application, the main thread (thread 0) of the program enqueues all the memory and compute tasks into the work queue, and sets up the dependency between tasks. Next, the main thread spawns one software thread for each processor core via Linux *Pthread*. We pin the threads to the cores using affinity to avoid high context-switching overheads. These child threads dequeue tasks from the work queue according to the proposed dynamic memory task scheduling policy. A lock and a counter are used to reinforce MTL restriction. We compile the synthetic workloads without optimization. While for dft, streamcluster and SIFT, we compile both the original and modified codes with the default compilation parameters specified by the workload vendors.

To show the performance benefit of the proposed run-time mechanism, we compare the speedup of the run-time throttling

[3]The *native* input size provided by the vendor of PARSEC benchmark suite is 128 dimensions.

mechanism with *Offline Exhaustive Search* and *Online Exhaustive Search*. The *Offline Exhaustive Search* policy chooses the best MTL value based on off-line runs. MTL is fixed throughout a program's execution. The *Online Exhaustive Search* policy dynamically adjusts *MTL* through a naive method without the help of the analytical model. This policy monitors the total execution time of W task pairs, and triggers *MTL* selection when the change in the execution time compared to previous W task pairs is larger than a pre-defined threshold. We test a wide range of threshold values, and the experimental results show that 10% gives the best performance for the workload tested in this work. To select MTL, the *Online Exhaustive Search* policy monitors the execution time of W task pairs under different MTL values, and then selects the one with the highest performance. By comparing with the *Online Exhaustive Search* policy, we can demonstrate the importance of utilizing the analytical model to detect phase change and select MTL in the proposed approach.

All the experiments are performed by root exclusively on Intel 2.80GHz Core i7-860 (Nehalem) with 8MB last-level cache[4]. We use a 2GB 1066MHz DDR3-SDRAM, which has one channel associated with two ranks. These two ranks are packaged in one DIMM, and each rank is composed of eight 1Gb chips. The width of the channel is 64 bits, and the bandwidth is 8.5GB/s. Many of the service routines on the system are disabled to reduce system noise. The execution time of entire workloads and memory/compute tasks (T_{m_k} and T_c) were measured by *gettimeofday()* function, which provides μsec accuracy for timing measurement. The smallest memory task in real-world benchmarks has at least 8K memory requests; therefore, the measurement done by gettimeofday() leads to negligible errors on the measurement of T_{m_k}. To further reduce the influence of system noises, we run each workload 20 times in sequence and average the results of the middle 10 runs (for corner case elimination) to represent the general program behaviors on the real machine.

To justify the quality of the modified workloads based on the stream programming model, we compare the modified codes with the original ones. Experiments show that, on average, our stream program codes have equivalent and better performance than the original code for sequential and multi-threaded runs, respectively.

VI. Experimental Result

In this section, we first analyze the performance speedup of the synthetic micro-benchmark with varied T_{m_1}/T_c ratios and memory footprints to observe the potential benefit of the interference-aware throttling and corroborate the proposed analytical model with the Intel i7 (Nehalem) machine. We then show the performance improvement provided by the dynamic memory thread throttling mechanism for real-world applications. Furthermore, we analyze the sensitivity of monitoring overhead with different number of tasks in the dynamic mechanism. To observe the benefit of dynamic adaptation of *MTL*, we also study the speedup of different phases in the application and the performance improvement of the same application with different input sets. Finally, we analyze the scalability of the dynamic throttling mechanism by doubling the available bandwidth and enabling SMT on the Intel i7 (Nehalem) machine. All the speedup results are obtained by comparing against the conventional interference-oblivious scheduling, where all the available threads and tasks are executed concurrently.

A. Corroboration of Analytical Model

To corroborate the analytical model with real hardware, we run synthetic workloads with large range of T_{m_1}/T_c on an Intel i7 quad-core machine. We vary the memory footprint of each memory task to observe the impact from capacity misses. For each synthetic workload, we run through the program from $MTL = 1$ to $MTL = n$ and report the highest speedup among different *MTL* constraints. We use *Static MTL* (*S-MTL*) to represent the *MTL* that leads to the highest performance improvement. The speedup reported by the real quad-core machine is compared against the speedup estimated by the analytical model.

Figure 13 shows the performance speedup of the synthetic workload with different memory footprints and a wide range of T_{m_1}/T_c ratios. We can see that the speedup estimated by the analytical model matches well with the speedup obtained from the quad-core machine. For this set of synthetic benchmarks, the speedup achieved by the interference-aware scheduling method can be up to 1.21x. The slightly inaccurate predictions at some micro-benchmarks come from the scheduling in non-steady states at the beginning and the end of the program.

From Figure 13, we can also observe that workloads with different T_{m_1}/T_c perform the best at different *MTL* (*S-MTL*) constraints. This also motivates us to develop the dynamic memory thread throttling mechanism which adjusts *MTL* dynamically to adapt to the workload variation. The synthetic workloads shown in Figure 13 are separated into three different regions according to their *S-MTL*. When $0.01 \leq T_{m_1}/T_c \leq 0.33$, *S-MTL* is equal to one. According to the analytical model, when $T_{m_1}/T_c \leq 0.33$, all cores are busy at $MTL \geq 1$ as shown in Figure 8(a). Therefore, in this case, *MTL=1* can achieve the best performance improvement by minimizing T_m without introducing unnecessary core idle time. That is, the best *MTL=1* derived by the analytical model matches the experimental results. When $T_{m_1}/T_c > 0.33$ (i.e., the second and third regions), according to the analytical model, *MTL=1* causes some cores to unnecessarily stay idle, that adversely impacts the overall performance as illustrated in Figure 8(a). In this case, the analytical model needs the values of T_{m_2}/T_c and T_{m_3}/T_c to predict the best *MTL*. The results show that the *MTL* selected by the analytical model matches the *S-MTL* values reported in Figure 13.

The hill-shape of the speedup in each region can be both derived by the analytical model and observed in the experimental results. According to the analytical model, the peak performance speedup at each region of workloads appears at the T_{m_k}/T_c equal to $\frac{k}{n-k}$ as shown in Equation 1. The reason is as follows. In the left hand side of each region where $T_{m_k}/T_c \leq \frac{k}{n-k}$ (i.e., all cores are busy at *MTL=k*), the performance speedup is equal to $\frac{T_{m_n}+T_c}{T_{m_k}+T_c}$

[4]Experiments on an Intel Q9550 quad-core machine with 12MB last-level cache also reveal similar speedup trend.

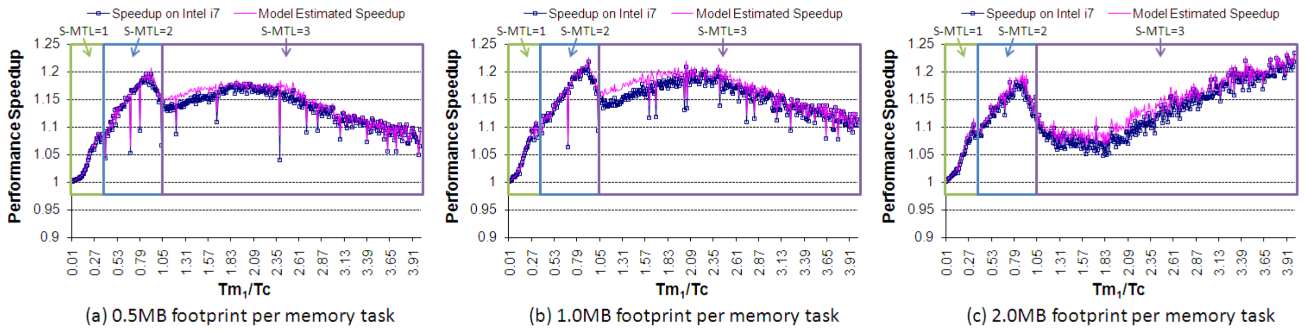


Figure 13: Performance speedup of synthetic workloads on the Intel i7 machine with a large range of T_{m_1}/T_c and various memory footprints per memory task: (a)0.5MB (b)1MB (c)2MB. *S-MTL* represents the *MTL* that leads to the highest speedup.

as shown in Figure 9(a). Since T_{m_n} and T_{m_k} are constant in the synthetic workloads, the speedup increases as T_{m_1}/T_c increases. As a result, we can observe a rising slope in the left hand side of each region. In the right hand side of each region where $T_{m_k}/T_c > \frac{k}{n-k}$, some cores are idle at *MTL=k* and Figure 9(b) shows that the performance speedup is equal to $\frac{(T_{m_n}+T_c)\times k}{T_{m_k}\times n}$. Since T_{m_n}, T_{m_k}, n and k are all constant, the speedup decreases as T_{m_1}/T_c increases. Therefore, we can observe a descending slope in the right hand side of each region. Due to space limitations, the proofs of the relations between T_{m_1}/T_c and the performance speedup are omitted. Both slopes together form a hill-shape at each region with distinctive *S-MTL*. The hill-shapes can be observed clearly in Figure 13(a) and (b). Note that there's no descending slope in the region of *S-MTL=3* in Figure 13(c). The reason is as follows. In the case of 2MB memory footprint of each memory task, instructions and data together exceed the capacity of the last-level cache. Therefore, compute tasks access memory due to capacity misses and interfere with the executions of memory tasks. These cases are not covered by the analytical model. Nonetheless, in the real-world workload experiments in the following, we always make sure the memory footprint of each memory task fits into the last-level cache following the stream programming model, where the analytical model sustains its accuracy.

B. *Performance Speedup of Run-Time Memory Thread Throttling*

To evaluate and analyze the performance improvement of the dynamic memory thread throttling mechanism, we observe the speedup of several real-world applications on the Intel i7 quad-core machine. Figure 14 shows the performance speedup of *dft*, *streamcluster*, and *SIFT* with dynamic memory thread throttling and compares to the best speedup by a *static MTL* assignment derived from offline searching *MTL* from 1 to 4. We also compare to the speedup derived from the naive online *MTL* searching algorithm, the *Online Exhaustive Search* policy described in Section V, to show the contribution of our analytical model in fast and adequate *MTL* selection with little monitoring overhead of T_{m_k} and T_c. We demonstrate only the results from the most adequate W (W represents the number of memory-compute task pairs that are monitored for T_{m_k} and T_c estimation) setting that leads to the highest performance speedup. The effect of different W setting is analyzed later in Section VI-C. We also use *D-MTL* to represent the *MTL* selected by dynamic policies.

The result shows that the dynamic memory thread throttling mechanism provides about 12% speedup on average and up to 20% speedup for *streamcluster*. Furthermore, the dynamic mechanism provides similar speedup compared to the best speedup by a static *MTL* assignment derived from offline exhaustive searching of *MTL*. Indeed, *D-MTL* outperforms the static assignment in *SIFT* since it is able to adapt to varied program phases in *SIFT*. We can also observe that the dynamic memory thread throttling mechanism provides about 5% higher speedup on average than the naive *MTL* selection method due to less monitoring overhead and better *MTL* selection. For example, the monitoring overhead of the proposed dynamic mechanism is about 0.04% of total execution time in *streamcluster*, while the overhead of *Online Exhaustive Search* is 4.87%. Furthermore, *Online Exhaustive Search* sometimes provides inaccurate performance estimation which leads to wrong *MTL* selection. The monitoring period of small number of tasks may not perfectly represent overall performance of the application due to the irregular scheduling overhead and the impact of load imbalance. In contrast, the proposed mechanism excludes the influence of unpredictable scheduling overhead and load imbalance by monitoring individual memory and compute tasks to estimate performance speedup at the steady state, thus results in better *MTL* selection and higher performance speedup than *Online Exhaustive Search*.

From Table II, we know that T_{m_1}/T_c of the *dft* kernel is equal to 12.77% (≤33%), which indicates that all cores are busy when $MTL \geq 1$ on the Intel i7 quad-core machine. Since the interference among memory requests decreases as MTL decreases, the dynamic memory thread throttling mechanism selects *D-MTL=1* eventually for the highest performance speedup. On the other hand, the T_{m_1}/T_c of *streamcluster* is equal to 37.14% (>33%), which indicates

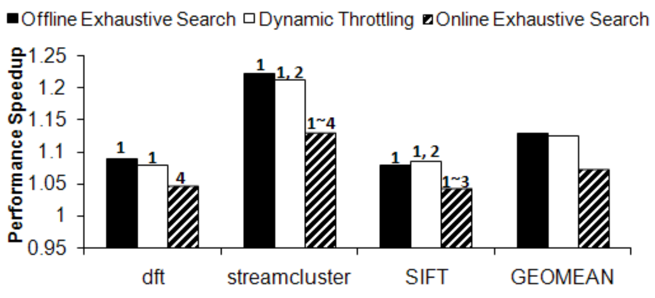


Figure 14: Speedup of realistic workloads on the Intel i7 machine with 4 threads scheduling. The number on each bar represents the selected *MTL*.

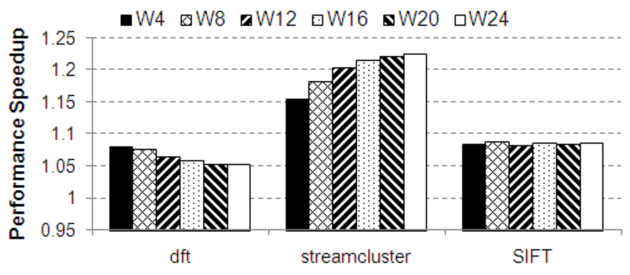


Figure 15: Speedup of applications with different W settings in the dynamic mechanism.

that some cores are idle at $MTL = 1$. Therefore, the dynamic mechanism selects the *MTL* value between 1 and 2, and provides 21.29% speedup. The performance speedup of *SIFT*, which consists of phases with different memory-to-compute ratios, is 8.58%. We will further analyze the adaptation of *MTL* for different phases in *SIFT* in Section VI-D1.

C. Sensitivity of Monitoring Overhead

As discussed above, W represents the number of memory-compute task pairs that are monitored for T_{m_k} and T_c estimation. To observe its performance impact of W, we vary the value of W from 4 to 24. We omit the experiments with $W > 24$, because the monitoring overhead of $W > 24$ adversely affects the overall performance significantly.

We compare the performance improvement from different W settings as shown in Figure 15. Higher values of W provide more accurate estimation of T_{m_k} and T_c. However, the monitoring overhead involved in searching adequate *MTL* is also large when W is high. This overhead becomes more obvious when number of parallel tasks in the application is relatively small. For example, the *dft* kernel has only 96 parallel memory-compute task pairs. Therefore, the monitoring overheads dominate the program execution and bring adverse effect on overall performance when $W > 8$. Apparently, the overhead of exhaustive search in *dft* is prohibitive. For *streamcluster* and *SIFT*, $W = 16$ is enough for accurate monitoring of T_{m_k} and T_c to select adequate *MTL* .

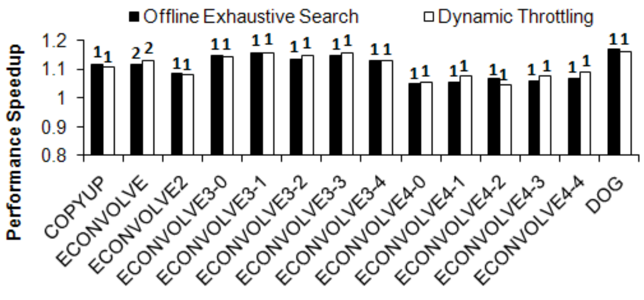


Figure 16: Speedup of main parallel functions in SIFT. The number on each bar represents the selected *MTL*.

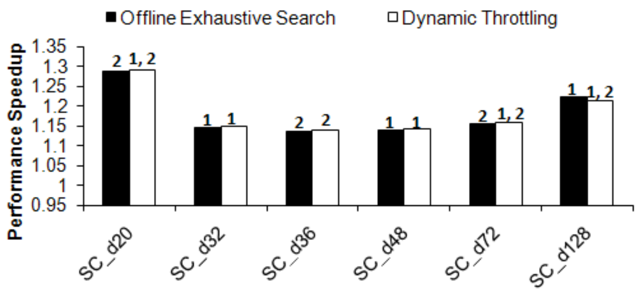


Figure 17: Speedup of *streamcluster* with different input settings. The number on each bar represents the selected *MTL*.

D. Effectiveness of Dynamic MTL Adaptation

In this subsection, we show that the proposed dynamic thread throttling mechanism provides an effective way to adapt *MTL* for different program characteristics.

1) MTL Adaptation to Program Phases: Figure 16 shows the speedup and *D-MTL* of main parallel functions in *SIFT*, compared to the speedup derived from the *Offline Exhaustive Search* policy. We observe that the dynamic mechanism selects different *D-MTL* for distinctive functions and provides performance improvement similar to *Offline Exhaustive Search*. For example, from Table III we know that T_{m_1}/T_c in the *ECONVOLVE* function is equal to 70.04% (>33%), which indicates that some cores are idle and adversely impact the overall performance when *MTL=1*. However, according to T_{m_2}/T_c measured in the experiments, we know that all cores are busy when $MTL \geq 2$. Therefore, the *MTL* selection policy chooses $MTL = 2$ as *D-MTL* after comparing against the performance estimated by the analytical model at $MTL = 1$ and $MTL = 2$. On the other hand, T_{m_1}/T_c in the *ECONVOLVE2* function is equal to 7.83% (≤33%), which indicates that all cores are busy when $MTL \geq 1$. Hence, the dynamic mechanism detects the phase change and switches *D-MTL* to 1. These results indicate that the dynamic memory thread throttling mechanism successfully detects phase changes and selects adequate *MTL* for each phase to reduce the interference among memory requests and improve system performance. Note that, the *MTL* values are the same for both *Offline Exhaustive Search* and the proposed dynamic approach. There are slight speedup differences, because the proposed dynamic approach executes the program with other *MTL*s to monitor T_m and T_c for *MTL* selection.

2) MTL Adaptation to Program Input Sets: Different input sets for the same application may vary the characteristic (i.e, the memory-to-compute ratio) of the program. Our proposed dynamic mechanism can intelligently choose different *MTL*s for these input sets to provide the highest performance speedup. Figure 17 shows the speedup of several instances of *streamcluster* with different input array dimensions. We can observe that the dynamic mechanism selects different *MTL*s for instances with different memory-to-compute ratios, and provides similar performance compared to *Offline Exhaustive Search*. For example, when the input array dimension equals to 32, *D-MTL=1* since all cores are busy at $MTL \geq 1$. On the other hand, *D-MTL=2* when the input array dimension equals to 36. From Table II, we know that T_{m_1}/T_c of *SC_ d36* is equal to 54.13% (>33%), which indicates that some cores are idle when *MTL=1*. After monitoring T_{m_2} and T_c, the dynamic mechanism finds that all cores are busy when *MTL=2*. Therefore, the dynamic mechanism chooses $MTL = 2$ for higher performance speedup after comparing against the performance estimated by the analytical model at $MTL = 1$ and $MTL = 2$.

E. Scalability Analysis of Dynamic MTL Adaptation

To evaluate the scalability of the proposed dynamic memory thread throttling mechanism, we run real-world workloads on a 2-DIMM system, which has 2 DDR3-SDRAM channels, and each channel connects to one DIMM with 2GB 1066MHz DDR3-SDRAM. The width of each channel is 64 bits and the total bandwidth of the system is 17GB/s. The left hand side of Figure 18 shows the speedup when running 4 threads. The results show that the dynamic memory thread throttling mechanism provides from 3.0% to 9.1% speedup on the 2-DIMM system. Similar to the 1-DIMM system presented above, the proposed mechanism achieves slightly higher speedups than the *Offline Exhaustive Search* approach in some cases. The speedup in a double-channel system is lower than in a single-channel system as expected since the parallelism between channels alleviates the interferences among memory requests. To stress the 2-DIMM memory system, we turn on the 2-way SMT capability of Intel i7. The results are shown in the right hand side of Figure 18. We can see that with more concurrent threads, the interference-aware scheduling shows more performance gain. For *streamcluster*, the speedup is 13.3%. Although our analytical model cannot accurately model the behavior of a SMT-enabled system since T_c is no longer a constant, the proposed dynamic memory thread throttling mechanism still performs quite well compared to the *Offline Exhaustive Search* approach.

VII. Related Work

To alleviate the memory request interferences among different threads, memory access scheduling policies in memory controller are proposed [10], [11], [22]. O. Mutlu et al. improve the system

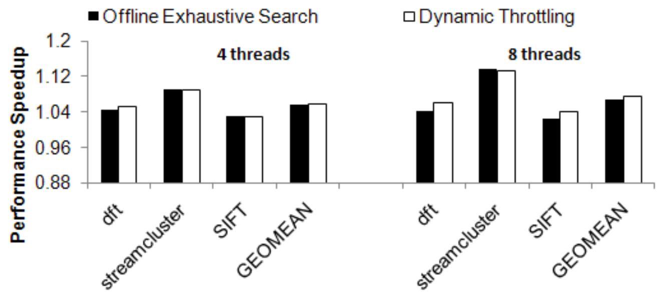


Figure 18: Speedup of realistic workloads on the Intel i7 machine without and with SMT enabled (4/8 threads scheduling) in a 2-DIMM system.

throughput by processing memory accesses from a thread in parallel [22]. Z. Fang et al. propose to exploit the locality in memory accesses from the same core [11]. E. Ebrahimi et al. propose a coordinated management [10] of all shared resources including off-chip bandwidth. However, our approach is to manage the memory tasks in a coarse-grained fashion to exploit the locality.

Besides the hardware approaches, software approaches are also proposed to ease the interferences among memory requests by reducing the bandwidth requirement [9], [18], [27] or managing the limited off-chip bandwidth for concurrently running threads [3], [5], [28]. C. Ding et al. propose a compilation technique [9] to improve cache reuse by improving memory locality. M. Karlsson et al. propose a run-ahead data fetching [18] to reduce the bandwidth consumption. C. D. Antonopoulos et al. propose to adjust the number of threads of concurrently running applications to fully utilize processors [3]. To consider contentions with other shared resources, R. Bitirgen et al. propose a machine learning method [5] to manage the shared resources in a coordinated fashion. S. Zhuravlev et al. propose a scheduling algorithm with comprehensive thread classification schemes [28] to reduce the contention in shared resources. D. Xu et al. try to keep the total bandwidth requirement at a steady level [27] to minimize performance degradation due to contention. These aforementioned methods only avoid contention when the bandwidth of the off-chip bus is saturated or nearly saturated. However, our approach interleaves memory tasks to alleviate the interferences among memory requests and minimize their latencies even when the bus bandwidth is not saturated.

Another approach to the interference problem among memory requests is the stream programming model, which keeps the intermediate results local to reduce off-chip accesses [24]. The well known stream architectures include Imagine [19], Merrimac [7], and Cell processor [17]. To simplify the use of stream architectures, stream programming languages [8], [12], [25], [26] are proposed. StreamIt [25] with some compilation optimizations [12] separates communication and computation to hide memory latency and models the producer-consumer locality to reduce memory requests. A. Das et al. propose to schedule a stream program with strip-mining, loop-unrolling and software-pipelining to optimize performance

[8] S. Liao et al. propose a data and computation transformation technique [26] to improve locality and minimize footprint, such that reducing memory requests mitigates the interferences.

VIII. Conclusion

In this paper, we propose a run-time memory thread throttling mechanism based on the idea of restricting the number of concurrent memory tasks to reduce the interference among memory requests and improve system performance on multi-core architecture. To better understand the performance speedup at different memory task scheduling constraints, we develop an analytical model to capture the essence of on-chip compute and off-chip communication. We corroborate the analytical model with experimental results from real hardware. Based on this analytical model, we design a policy to detect phase change in applications and intelligently select adequate scheduling policy for each phase to tackle the memory wall problem. Our results show up to 20% speedup with a pool of synthetic workloads. For realistic benchmarks, our run-time mechanism shows 12% speedup on the Intel i7 quad-core machine. Sensitivity studies of the proposed mechanism with varied monitoring time, input working sets, program phases and hardware configurations show that the proposed run-time mechanism is robust and exhibits equivalent performance to a mechanism based on offline exhaustive search. Moreover, the proposed mechanism provides 5% higher speedup on average than a naive method based on online exhaustive search. The higher speedup of the proposed mechanism comes from less run-time monitoring overhead and better selection of the scheduling policy for each phase. We are currently working on extending the scalability study in this paper to an IBM POWER7 machine that has substantially more hardware threads than the Intel i7-based systems.

IX. Acknowledgments

This research is supported in part by research grants from ROC National Science Council NSC 99-2220-E-002-027-, NSC 99-2000-E-002-021, NSC 98-2911-I-002-077, Excellent Research Projects of National Taiwan University 99R80304, Macronix International Co., LTD. 99-S-C25 and IBM Faculty Award. The authors would like to thank Ramakrishnan Rajamony, Kevin Nowka, Sani Nassif, Elmootaz-bellah N. Elnozahy and Yuan Xie for their coordination and support. Pen-Chun Yew, Wei-Chung Hsu, Chung-Ho Chen, Hsien-Hsin S. Lee, and Eiman Ebrahimi have offered helpful insights to refine the early draft of this work. We also appreciate the insightful comments and constructive suggestions from the anonymous reviewers.

References

[1] OpenCV. http://opencv.willowgarage.com/wiki/.
[2] SIFT++. http://www.vlfeat.org/~vedaldi/code/siftpp.html.
[3] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Realistic Workload Scheduling Policies for Taming the Memory Bandwidth Bottleneck of SMPs. In *Proceedings of the 11th Annual International Conference on High Performance Computing*, December 2004.
[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.
[5] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. In *Proceedings of the 41st Annual ACM/IEEE International Symposium on Microarchitecture*, November 2008.
[6] D. Burger, J. R. Goodman, and A. Kägi. Memory Bandwidth Limitations of Future Microprocessors. In *Proceedings of the 23rd annual International Symposium on Computer Architecture*, May 1996.
[7] W. J. Dally, P. Hanrahan, M. Erez, and T. J. Knight. Merrimac: Supercomputing with Streams. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, November 2003.
[8] A. Das, W. J. Dally, and P. Mattson. Compiling for Stream Processing. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, September 2006.
[9] C. Ding and K. Kennedy. Improving Effective Bandwidth through Compiler Enhancement of Global Cache Reuse. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, April 2001.
[10] 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 *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2010.
[11] Z. Fang, X.-H. Sun, Y. Chen, and S. Byna. Core-aware Memory Access Scheduling Schemes. In *Proceedings of the 23th IEEE International Parallel and Distributed Processing Symposium*, May 2009.
[12] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.
[13] J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum. Streamware: Programming Gerneral-Purpose Multicore Processors Using Streams. In *Proceedings of the 13th international conference on Architectural Support for Programming Languages and Operating Systems*, March 2008.
[14] J. Gummaraju, M. Erez, J. Coburn, M. Rosenblum, and W. J. Dally. Architectural Support for the Stream Execution Model on General-Purpose Processors. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, September 2007.
[15] J. Gummaraju and M. Rosenblum. Stream Programming on General-Purpose Processors. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, November 2005.
[16] J. L. Hennessy and D. A.Patterson. Computer architecture: a quantitative approach. 2006.
[17] H. P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, February 2005.
[18] M. Karlsson and E. Hagersten. Conserving Memory Bandwidth in Chip Multiprocessors with Runahead Execution. In *Proceedings of the 21th International Parallel and Distributed Processing Symposium*, March 2007.
[19] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media Processing with Streams. In *IEEE MICRO*, March-April 2001.
[20] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, 2004.
[21] N. R. Mahapatra and B. Venkatrao. The processor-memory bottleneck: problems and solutions. *Crossroads*, 5(3es):2, 1999.
[22] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *Proceedings of the 35th annual International Symposium on Computer Architecture*, June 2008.
[23] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, September 2007.
[24] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens. A Bandwidth-Efficient Architecture for Media Processing. In *Proceedings of the 31th annual IEEE/ACM International Symposium on Microarchitecture*, November-December 1998.
[25] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, April 2002.
[26] S. wei Liao, Z. Du, G. Wu, and G.-Y. Lueh. Data and Computation Transformations for Brook Streaming Applications on Multiprocessors. In *Proceedings of the 4th Annual International Symposium on Code Generation and Optimization*, March 2006.
[27] D. Xu, C. Wu, and P.-C. Yew. On Mitigating Memory Bandwidth Contention through Bandwidth-Aware Scheduling. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, September 2010.
[28] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2010.