

# Modeling Optimistic Concurrency using Quantitative Dependence Analysis

Christoph von Praun    Rajesh Bordawekar    Călin Cașcaval

IBM T.J. Watson Research Center  
{praun, bordaw, cascaval}@us.ibm.com

## Abstract

This work presents a quantitative approach to analyze parallelization opportunities in programs with irregular memory access where potential data dependences mask available parallelism. The model captures data and causal dependencies among critical sections as algorithmic properties and quantifies them as a density computed over the number of executed instructions. The model abstracts from runtime aspects such as scheduling, the number of threads, and concurrency control used in a particular parallelization.

We illustrate the model on several applications requiring ordered and unordered execution of critical sections. We describe a run-time tool that computes the dependence densities from a deterministic single-threaded program execution. This density metric provides insights into the potential for optimistic parallelization, opportunities for algorithmic scheduling, and performance defects due to synchronization bottlenecks.

Based on the results of our analysis, we classify applications into three categories with low, medium, and high dependence densities. Applications with low dependence density are naturally good candidates for optimistic concurrency, applications with medium density may require a scheduler that is aware of the algorithmic dependencies for optimistic concurrency to be effective, and applications with high dependence density may not be suitable for parallelization.

**Categories and Subject Descriptors** D.1.3 [Software]: Concurrent Programming; D.2.3 [Software]: Coding Tools and Techniques; D.2.8 [Software]: Metrics

**General Terms** Measurement, Performance

**Keywords** available parallelism, dependence analysis, dependence density, implicit parallelism, optimistic concurrency, program parallelization, transactional memory

## 1. Introduction

Until recently parallel programming has been restricted to a small set of skilled practitioners who have the knowledge and expertise to extract the maximum parallelism from applications. With the current trends in computer system design, there is a need to make parallel programming more accessible. Even assuming that a parallel

algorithm is given for a particular application, coding that algorithm is not a trivial matter. The programmer must take care of data sharing, communication, and synchronization between tasks. These tasks are typically error prone and hard to debug.

Taking all these factors into consideration, we believe that simplifying parallel programming means not only providing new programming paradigms but also tools that allow programmers to understand concurrent behavior. These tools must help users focus on the important sections of an application, and provide guidance with respect to potential opportunities for optimization.

In this paper, we focus on abstractions that enable the construction of such tools in the context of optimistic concurrency such as provided by Transactional Memory (TM). TM is considered one of the paradigms that will simplify parallel programming. Previous work [6, 8, 20] argues, mainly qualitatively, about TM's benefits compared to other forms of synchronization, such as locks. These benefits are in terms of ease of programming, such as absence of deadlocks, composability, local reasoning (avoiding lock management), etc., and in terms of performance, such as optimistic concurrency. Most of the previous work on TM evaluations has focused on different designs and presented results with respect to the specific features proposed. In addition, researchers have studied the parallelism available in applications on ideal parallel systems, such as [25]. As far as we know, this is the first study that abstracts a TM system and provides a quantitative study of application concurrency and synchronization characteristics.

We first define a program execution model that allows us to capture different forms of parallel execution: transactional optimistic concurrency [11] and optimistic parallelization, such as thread level speculation (TLS) [12]. We then derive two data dependence based metrics to quantify the potential contention probability of a task (block of code that can execute in parallel). Given these metrics, we discuss the amount of parallelism available to an application. In a nutshell, the parallel execution of the application can benefit if a parallel scheduler that has knowledge of data dependencies between tasks can decide on several execution modes: sequential, optimistically parallel, or truly parallel. In addition, optimistic parallelism can be supported by a TLS or TM system, or an inspector-executor paradigm [27], where dependencies are dynamically checked before the execution of the parallel section.

We apply this model to several applications from different domains: scientific (UMT2K and SSCA#2) [30, 1], bio-informatics (genome) [20], data mining (kmeans) [23], transactional processing and databases (vacation and MySQL keycache) [20, 22], and computational geometry (Delaunay) [13]. Based on their characteristics with respect to the metrics in our model, we classify these applications into three categories, namely those with low, medium, and high dependence density. We argue that the performance of applications with low dependence density can be enhanced by optimistic concurrency. Applications that have a medium dependence

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'08 February 20–23, 2008, Salt Lake City, Utah, USA.  
Copyright © 2008 ACM 978-1-59593-960-9/08/0002...\$5.00

density can benefit from optimistic concurrency with algorithmic modifications and the support of an informed scheduler. Finally, for application with a high dependence density, it is very difficult or impossible to benefit from current techniques of optimistic concurrency.

To summarize, this paper makes the following contributions:

- Defines an abstract model for parallelism based on the density of dependencies of tasks in an application, independent of the number and interleaving of execution threads.
- Presents a detailed, quantitative analysis of several applications that have been used to characterize different TM systems; we discuss the properties of these applications and provide insights in their behavior with respect to optimistic synchronization and parallelism.
- Describes a tool that incorporates our model and produces the analysis on-the-fly based on a single-threaded program execution.

The rest of the paper is organized as follow: Section 2 introduces our abstract model of parallelism, Section 3 presents our tools and experimental setup, and Section 4 discusses the application characterization. We discuss related work in Section 5 and conclude in Section 6.

## 2. Abstract model

We first define an abstract model of program execution to capture data dependencies. The model identifies in a program distinct *phases* that define the set of parallel tasks that can execute concurrently at any point in time. The model can represent parallel programs with both explicit and implicit parallelism. In explicit parallel programs, accesses to shared data are protected by critical sections. Implicit parallel programs use optimistic parallelism and rely on the system to detect conflicting accesses.

### 2.1 Execution model

A *program execution* consist of one or several sequences of executed instructions (threads) (Figure 1a). In our abstract model, we consider inter-dependencies between sets of tasks (defined below) from any such sequence of instructions and rely on a scheduler to pick any tasks such that it maximizes concurrency. The thread of execution from where the task originates is irrelevant and will be ignored in the subsequent discussion.

**Tasks** A task is an atomic unit of execution, i.e., a sequence of instructions that execute serially and are divided into further concurrent units. Also, a task is the smallest unit of independent work that may be executed concurrently with other tasks (Figure 1b).

For the dependence analysis model we consider only in the execution of *critical sections* and *optimistic parallel regions*. Nested critical sections are assumed flattened. Critical sections and optimistic parallel regions, are defined statically as program constructs. Dynamic instances of a critical section in a program execution are referred to as *unordered tasks*, and a dynamic instance of a speculatively parallel region is an *ordered task*. A task may be embedded into a computational context that occurs outside a critical section; the model ignores such auxiliary computation since it does not contribute to inter-task dependencies or constrain the scheduling of tasks.

**Phases** A program execution can be partitioned into a sequence of *phases* (Figure 1a). Examples of program constructs that define the execution in phases are loops with speculative parallelization or fork-join regions in an explicit parallel program. The concept of a phase enables the dependence analysis to consider only a specific part of the execution and to exclude known parallel or inherently

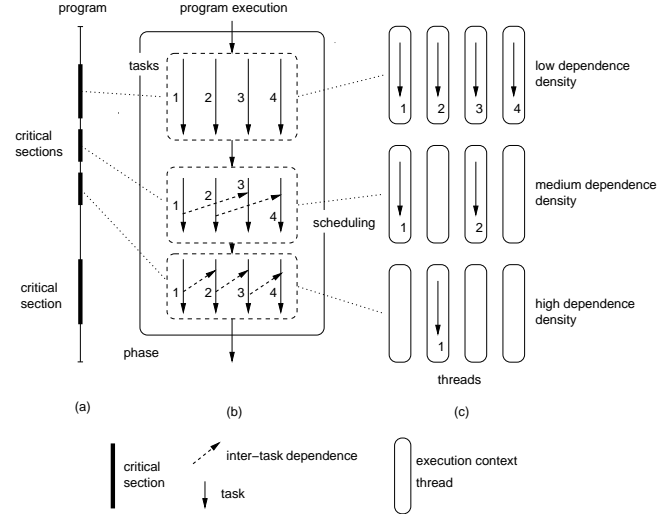


Figure 1. Abstract execution model.

serial parts of a program. The dependence analysis is applied independently to each program phase. In this abstract model, a phase consists of a set of tasks; we assume (by construction) that tasks in a phase are either all *ordered* or all *unordered*.

**Inter-task dependencies** The main limiting factor of concurrent task execution are *memory-level dependencies*. A memory-level dependence exists between tasks  $t_1$  and  $t_2$  if the tasks access a shared variable and one of the tasks writes (Figure 1b). A memory-level dependence creates an inter-task dependence. For example, a memory-level dependence may manifest itself as a conflict in transactional memory when dependent tasks are executed as concurrent transactions.

Applications with ordered tasks specify for each memory-level dependence the order in which the accesses should occur. This is typically done by imposing the ordering between tasks to the memory operations. The order can be a total order, as is commonly assumed with optimistic loop parallelization, or a partial order, as described in Section 4 in the case of *umt2k*. We refer to this high-level order specified by the programmer as *algorithmic order*. The algorithmic order induces transitive dependency chains across tasks which we call *causal dependence*.

**Scheduler** The role of a *scheduler* is to achieve concurrency by mapping tasks onto parallel threads under consideration of algorithmic dependencies between tasks (Figure 1c). A scheduler may, but does not necessarily have to, take into account information about memory-level dependencies when making scheduling decisions. If a scheduler chooses to ignore memory-level dependencies then we assume that appropriate concurrency control mechanisms are in place at run time to guard against concurrent conflicting memory access, i.e., violations of memory-level dependencies. We shall discuss how the choice of a schedule affects the available parallelism in the application in the next sections.

### 2.2 Dependence density

In this section, we introduce the notion of *dependence density* to characterize the inter-task dependencies. Intuitively, the *dependence density* expresses the probability of existing memory-level dependencies among any two randomly chosen tasks from the same program phase. We use this metric to quantify the available parallelism in a phase with unordered tasks.

In the following, we devise different dependence density metrics for unordered tasks (*data* dependence density) and ordered tasks (*causal* dependence density) because the nature of dependencies changes when tasks are ordered. These metrics express the potential concurrency and the 'reward' that a scheduler may reap when considering memory-level dependencies.

To formally define dependence densities, we first define the read and write sets of a task  $t$  as follows:

$$read\_set(t) := \{l \mid t \text{ reads from location } l \text{ and has not previously written to location } l\}$$

$$write\_set(t) := \{l \mid t \text{ writes to location } l\}$$

A data dependence between tasks  $t_1$  and  $t_2$  is defined as:

$$flow\_dep(t_1, t_2) := write\_set(t_1) \cap read\_set(t_2)$$

$$has\_data\_dep(t_1, t_2) := t_1 \neq t_2 \wedge (flow\_dep(t_1, t_2) \neq \emptyset \vee flow\_dep(t_2, t_1) \neq \emptyset)$$

This definition ignores write-write dependencies, because such dependencies do not necessarily limit concurrency, but rather just introduce potential data races.

The data dependence density,  $data\_dep\_dens(t)$ , for a task  $t$  expresses the fraction of tasks (measured using their execution length) with which  $t$  has a data dependence. Let  $T_p$  be the set of unordered tasks in phase  $p$ , and  $T$  the set of tasks considered in our analysis. Let  $len(t)$  be the size of a task  $t$ , e.g., the number of cycles or instructions.

$$data\_dep\_dens(t) := \frac{\sum_{s \in T_p \wedge has\_data\_dep(t, s)} len(s)}{\sum_{t \in T_p - \{t\}} len(t)}$$

$$data\_dep\_dens(T) := \frac{\sum_{t \in T} data\_dep\_dens(t)}{|T|}$$

Note that the dependence density for an individual task  $data\_dep\_dens(t)$  is always computed over the set  $T_p$ , since the model regards any pair of tasks as potentially concurrent. The data dependence density for a set of tasks  $T$  is the average of the dependence densities of the individual tasks in  $T$ .  $T$  can be chosen as either all tasks in a phase or a subset, such as all the dynamic instances of a particular task. In the evaluation (Section 4), e.g., the dependence densities are computed for the sets of dynamic task instances corresponding to critical sections in the program code.

The values of the dependence density fall into the interval  $[0, 1]$  – lower numbers indicate a lower probability of a memory-level dependence violation when tasks execute concurrently, i.e., higher degrees of potential parallelism.

### 2.3 Causal dependence density

Similar to the data dependence density, we introduce the causal dependence density to characterize inter-task dependencies for ordered tasks. Causal dependence imposes the following additional constraint: an execution schedule of the tasks must adhere to data dependence chains that may transitively spawn across a sequence of tasks. We call the inter-task dependencies resulting from data dependence and task ordering *causal dependence*.

Let read and write sets of a task  $t$  be defined as in Section 2.2. We specify the algorithmic order as sets of predecessor and successors tasks of a task  $t$  as follows:

$$pred(t) := \{s \mid \text{execution of task } s \text{ must precede } t\}$$

$$succ(t) := \{s \mid \text{execution of task } s \text{ must follow } t\}$$

The *pred* and *succ* sets do not necessarily define a total order among tasks. A task  $t_1$  has a causal dependence with some task  $t_2$  if either of the following cases holds: task  $t_2$  is a predecessor of  $t_1$  and there exists a chain of dependencies from a write of  $t_2$  to a read of  $t_1$ , or alternatively, task  $t_2$  is a successor of  $t_1$  and there exists a dependence chain from a write of  $t_1$  to a read of  $t_2$ .

We formally define this relationship as follows:

$$has\_causal\_dep(t_1, t_2) := (t_2 \in succ(t_1) \wedge has\_causal\_chain(t_1, t_2)) \vee (t_2 \in pred(t_1) \wedge has\_causal\_chain(t_2, t_1))$$

We describe two variants of determining causal dependence chains, *has\\_causal\\_chain*. First, predicate *has\\_causal\\_chain\\_precise*( $t_1, t_2$ ) specifies a precise computation of causal dependence chains.

$$has\_causal\_chain\_precise(t_1, t_2) := (\exists l \forall t_3 \cdot l \in flow\_dep(t_1, t_2) \wedge t_3 \in succ(t_1) \cap pred(t_2) \wedge l \notin write\_set(t_3)) \vee (\exists t_3 \cdot t_3 \in succ(t_1) \cap pred(t_2) \wedge has\_causal\_chain\_precise(t_1, t_3) \wedge has\_causal\_chain\_precise(t_3, t_2))$$

A causal dependence from task  $t_1$  to  $t_2$  exists if (i) the tasks communicate through some location  $l$  due to flow dependence and no tasks intervening between  $t_1$  and  $t_2$  in the algorithmic order breaks this communication, or (ii) some intermediate task  $t_3$  transitively establishes a causal dependence due to *has\\_causal\\_chain\\_precise*( $t_1, t_3$ ) and *has\\_causal\\_chain\\_precise*( $t_3, t_2$ ). We call this variant 'precise' since the dependence between two tasks  $t_1, t_2$  is determined independently of the results of the dependence analysis of tasks that intervene  $t_1$  and  $t_2$  in the algorithmic order. Thus this predicate has the potential to unveil causal independence among tasks that are 'distant' in some original sequential execution.

Second, predicate *has\\_causal\\_chain\\_seq*( $t_1, t_2$ ) computes an approximation of *has\\_causal\\_chain\\_precise*( $t_1, t_2$ ), as follows:

$$has\_causal\_chain\_seq(t_1, t_2) := flow\_dep(t_1, t_2) \neq \emptyset \vee \exists t_3 \in succ(t_1) \cap pred(t_2) \cdot flow\_dep(t_1, t_3) \neq \emptyset$$

Starting from a task  $t$ , all tasks *succ*( $t$ ) with which  $t$  has a flow dependence in some sequential execution, are considered causally dependent. This predicate provides a conservative over-approximation of the real causal dependence chains. However, it characterizes the behavior of schedulers that choose a task based on local knowledge and do not significantly depart from the same sequential task sequence when computing the schedule, i.e., the scheduler considers only the successor or a small window of tasks around task  $t$  in the sequential task sequence.

Note that the dependence chain across tasks may involve more than one variable, e.g.:  $write\_set(t_1) = \{x\}$ ,  $read\_set(t_3) = \{x\}$ ,  $write\_set(t_3) = \{y\}$ ,  $read\_set(t_2) = \{y\}$  results in a dependence chain  $t_1 \rightarrow t_3 \rightarrow t_2$ , hence *has\\_causal\\_chain*-(*seq|precise*)( $t_1, t_2$ ).

If not relevant, we omit to specify the variant used to compute the causal relation in the following and just refer to predicate *has\\_causal\\_chain*; in this case either of the variants *seq* or *precise* is a valid implementation.

Dual to the data dependence density, we define the density of causal dependencies of an individual task  $t$  and for a set of tasks  $T$  as follows:

$$\begin{aligned} \text{causal\_dep\_dens}(t) &:= \frac{\sum_{s \in T_p \wedge \text{has\_causal\_dep}(t,s)} \text{len}(s)}{\sum_{t \in T_p - \{t\}} \text{len}(t)} \\ \text{causal\_dep\_dens}(T) &:= \frac{\sum_{t \in T} \text{causal\_dep\_dens}(t)}{|T|} \end{aligned}$$

## 2.4 Available parallelism

Let  $T_p$  be the set of tasks in phase  $p$ . The dependence density defines the probability of conflict between a pair of tasks that a scheduler chooses from  $T_p$ . Given the dependence density for all pairs of tasks in a phase, we want a metric to capture the potential concurrency for the set of tasks.

The *available parallelism* metric captures this program property independently of the execution environment, i.e., of the number of parallel threads. The available parallelism is not necessarily equal to the possible speedup obtained for the phase. Several additional factors affect speedup: first, tasks may account only for a small fraction of the overall execution of a phase (coverage). Second, tasks may have different characteristics in the same phase (e.g., tasks corresponding to different critical sections) and produce load imbalance, thus wasting resources. Third, this model for available parallelism takes into account only the number of available threads and abstracts from other finite resources like memory bandwidth, latency, and cache operation that impact the achievable speedup. A complete model that accounts for all these factors would be significantly more complex, and would tie the abstract model to a particular machine configuration. For the scope of this paper, we strive to maintain machine independence and characterize the algorithmically available parallelism.

In the remainder of this section, we discuss scheduling decisions which affect the two dependence density metrics and available parallelism metrics.

**Un-informed scheduler** First, consider an *un-informed scheduler*. For unordered tasks, this scheduler randomly selects for execution any task; the probability of an inter-task dependence is modeled by the *data\_dep\_dens* metric. For ordered tasks, the execution order must be compatible with the algorithmic order; the probability of an inter-task dependence is captured by *causal\_dep\_dens*. In the following we use *dep\_dens* to mean one or the other metrics, depending on the ordering of the set of tasks.

Let  $n$  be the number of threads onto which the scheduler maps the execution. The available parallelism among tasks in  $T_p$  is calculated as follows:

$$\begin{aligned} \text{avail\_par}(T_p, n) &:= \sum_{k=0}^{n-1} (1 - \text{dep\_dens}(T_p))^k \\ &:= \frac{1 - (1 - \text{dep\_dens}(T_p))^n}{\text{dep\_dens}(T_p)} \\ \text{avail\_par}(T_p) &:= \lim_{n \rightarrow \infty} \text{avail\_par}(T_p, n) \\ &= \frac{1}{\text{dep\_dens}(T_p)} \end{aligned}$$

In this geometric series, the  $k$ -th term represents the amount of 'useful' work that can be achieved, on average, on the  $k$  processor under the assumption of homogeneous task size, and a pairwise

conflict probability of  $\text{dep\_dens}(T_p)$  with threads 0 to  $k - 1$ . The geometric series reflects the fact that, with non-zero dependence density, additional computational (growing number of threads) resources become less and less effective in returning performance increases. Notice that we make idealized assumptions about conflict management in this model, in that the addition of the  $k$ -th thread does not perturb or increase conflict probabilities in the previous  $k - 1$  threads. This is a reflection of our assumption that conflict probabilities are independent. This assumption holds by definition in the unordered case, and by construction in the ordered case, since we use the causal dependence density metric.

As an example, a program phase with data dependence density of 0.1 under an un-informed scheduler results in an available parallelism of 5.69 with 8 threads and a maximum available parallelism of 10 with an unlimited number of threads.

**Algorithmic scheduler** The available parallelism computed according to the un-informed scheduler model may underestimate the parallelism that may be obtained by an *algorithmic scheduler*, i.e., a scheduler that chooses tasks according to the likelihood of interference with concurrently executing tasks. The benefit of such algorithmic scheduling has been demonstrated in [14] on the example of the Delaunay mesh refinement for unordered tasks. We will also discuss the effect of algorithmic scheduling for umt2k with ordered tasks in Section 4.

However, since the operation of an algorithmic scheduler is highly dependent on the data structure and algorithm, we do not give a formula on the basis of the current dependence analysis framework. Instead we discuss how the data dependence density metric can be intuitively used to estimate the available parallelism and present concrete results when we discuss applications that can benefit from such scheduling decisions.

There exist, of course, pathological cases, in which a workload will not benefit from algorithmic scheduling. Consider the case of an execution phase where a transaction reads and writes exactly one out of a total of two shared variables with equal probability. An algorithmic scheduler will not be able to do better than an un-informed scheduler, which achieves a maximum available parallelism of 2. The following is a model that captures such pathological and distinguishes it from cases such as Delaunay, which has potential for algorithmic scheduling.

The key idea is as follows: assume a dependence density of  $d = \text{dep\_dens}(T)$ . Let  $S \subset T$  be  $1/d$  randomly chosen tasks that are pairwise independent. Let  $S' \subset T$  be the set of tasks, independent of  $S$ , that conflict with some tasks in  $S$ . If  $|S'| \approx |S|$  and  $|S| + |S'| \ll |T|$ , then an algorithmic scheduler has ample room to choose tasks that are independent of those already in  $S$  and hence high potential to grow available parallelism. If  $|S'| \approx |T|$  then there is little potential for an informed scheduler to grow the available parallelism beyond  $1/d$ . The pathological case presented above has  $|S| = |S'| = 1$  and  $|T| = 2$ .

To capture this idea in the dependence analysis framework, we extend the definition of dependence density to sets of tasks: for a task  $t$  and a set of tasks  $S$ , let

$$\text{has\_dep}(t, S) = \exists s \in S \cdot \text{has\_dep}(t, s),$$

where *has\_dep* may stand for causal or data dependence. For a set  $S \subset T$  of  $k$  (pairwise) independent tasks, we define the *k-order dependence density* to the task length:

$$\text{dep\_dens}_k(S) = \frac{\sum_{t \in T_p - S \wedge \text{has\_dep}(t, S)} \text{len}(t)}{\sum_{t \in T_p - S} \text{len}(t)}$$

The concept of the  $k$ -order dependence density is applicable to data and causal dependence. The values fall into the interval  $[0, 1]$  and express the fraction of tasks in  $T_p$  that are dependent with some task in set  $S$ . Since  $|S| = k$ , a low value of the  $k$ -order dependence density indicates that there is room for an algorithmic scheduler to achieve parallelism beyond  $k$ .

## 2.5 Dependence granularity

The discussion until now focused on dependences between individual accesses. However, in many systems, conflict detection is performed at cache line granularity. To model such cases, we increase the granularity at which accesses are considered to cause dependences, based on their mapping into cache lines. The formulation remains the same, but computed dependence density will depend on the granularity. In Section 4.4 we evaluate the sensitivity of the applications to this metric.

## 2.6 Phase classification

Given the metric of available parallelism  $avail\_par(T_p)$  in an application, and taking into account the number of parallel threads  $n$  targeted by the scheduler, we introduce following classification of program phases (Figure 1c) according to the dependence density among tasks in that phase:

**high dependence density** :  $avail\_par(T_p) \ll n$ . The parallelism in such workload is inherently limited and the choice of scheduler has little or no impact on the available parallelism.

**medium dependence density** :  $avail\_par(T_p) \approx n$ . This workload is amenable to parallelization. The choice of scheduler can have significant impact on the available parallelism.

**low dependence density** :  $avail\_par(T_p) \gg n$ . Parallelism is ubiquitous and an un-informed scheduler can reach a high levels of parallelism, thus effectively exploit multithreading.

For an application, one can use this classification of program phases to estimate its parallel behavior and determine requirements for a task scheduler.

# 3. Methodology

## 3.1 Task profiler tool

We developed a profiling tool that records and evaluates the execution of program phases and tasks within these phases. The tool is an online dynamic binary instrumentation and tracing tool, based on the PIN [18] framework.

The tool operates on a single or multithreaded execution of an input program. The input program communicates events like the start and end of a program phase, task boundaries and ordering properties to the tool through marker calls. Currently, marker calls are inserted by the programmer but such information can easily be synthesized by a compiler from the critical regions and annotations for regions of speculation boundaries in the program source code.

At run time, the tool records properties of the program execution such as instruction count and memory accesses; moreover, the tool computes the dependence density, i.e., the data dependence density for phases with unordered tasks and the causal dependence density for phases with ordered tasks.

The dependence analysis as described in Section 2 considers memory access information for all tasks within a phase; also the density metric for an individual task is computed by pairwise dependence detection with all other tasks in phase  $p$ , which is of complexity  $O(|T_p|^2)$ .

The programs we target have typically thousand or more tasks within a phase and hence an exhaustive recording of per task information is impractical. Instead the task profiler samples tasks and

computes dependence information from the samples. The fraction of tasks sampled in a phase is a parameter to the tool. For phases with ordered tasks, a sample is a contiguous task sequence of a specified length. The causal dependence analysis is computed individually for each sample, i.e., the length of the sample provided the window size at which the analysis determines dependencies. For phases with unordered tasks, a sample consist of an individual task. The dependence analysis is performed for all tasks samples within a phase.

## 3.2 Experimental setup

We have conducted the dependence analysis on a set of applications that have been recently used to evaluate transactional memory systems [26, 32, 20, 15]. For this study, a phase usually covers the entire execution of the (speculatively) parallel region in the program. Unless specifically noted, applications are configured to execute in a single thread with the same input and command line arguments used in prior studies.

Since a single-threaded application run is used for profiling, data structures allocated in thread-specific memory could introduce artificial dependencies among tasks. Hence, we exclude certain locations in thread-specific storage from the read and write sets of tasks. Examples of thread-specific storage include the memory allocator and the random number generator in the STAMP benchmarks [20]. Also, we exclude automatic variables (typically allocated on the stack) from the read and write sets, since these are usually task-private.

For applications with ordered tasks, 2% of the tasks are sampled and each sample consists of 250 consecutive tasks. For applications with unordered tasks, we sampled 5% of the tasks; lower or higher sampling rates minimally affected the results.

The task profiling tool records read and write sets and detects dependencies. the tool reports the source code location (i.e., file name and line number), the fraction of the execution of the phase spent in tasks, (*coverage*) of this category, the number of dynamic task instances, (*count*), the number of instructions in a critical section, (*size*, the size of read and write sets (average per task and total per category of task), and the dependence density. The evaluation results (Section 4) reports these attributes as columns in Table 1 to Table 7. Read and write sets contain all addresses of static global and heap-allocated data (not stack addresses), irrespective of the sharing properties of those data. Unless noted otherwise, the set for conflict detection contain word-aligned addresses. The tool provides flexibility to choose different alignments and hence evaluate the effects of false sharing on the dependence density.

As mentioned in Section 3, the set of tasks to compute dependence densities can be chosen in different ways. For these experiments, we choose two such sets:

- dependence density relative to  $T_p$ , where  $T_p$  is the set of all tasks in the same program phase  $p$ ; this metric is reported in column *dep-density - all* in Table 1 to Table 7.
- dependence density relative to  $T_c$ , where  $T_c$  is the set of tasks corresponding to the same static instance of a critical section  $c$  or a parallel loop; this metric is reported in column *dep-density - same*.

For programs with ordered tasks, the causal dependence is computed for all tasks in the phase with the precise (*causal\_chain\_precise()*) and the approximated predicate (*causal\_chain\_seq()*). Note that even the precise predicate is not perfect since the dependencies are computed from a sampling window of 250 consecutive tasks. Both metrics for causal dependence density are reported in columns *dep-density - precise* and *dep-density - seq* in Table 6.

## 4. Evaluation

We experimented with the following applications: genome, kmeans and vacation from the STAMP suite [20], delaunay – a reimplementation of [13], SSCA#2 [1], UMT2K [30], and MySQL [22]. We categorized these applications according to the classification in Section 2.6, assuming an execution platform with few tens of threads.

### 4.1 Low dependence density

As mentioned before, benchmarks with a low dependence density can reach high levels of parallelism relatively easy, even with an un-informed scheduler.

#### 4.1.1 delaunay

The Delaunay triangulation is an algorithm for discretizing a two-dimensional domain into a mesh of triangles with certain quality guaranties. Mesh generation is an important aspect of graphics rendering and for solving finite-element solutions of partial differential equations. The Delaunay method refines a coarse initial mesh through iteration. The sequential algorithm repeatedly looks for a 'bad' mesh element that does not satisfy the quality constraints; computes a region around the 'bad' element, called its 'cavity', and replaces elements in the cavity by new elements. This process is called the refinement. Some of these new elements may not satisfy the quality constraints themselves. However, it can be shown that the algorithm always terminates and produces a guaranteed quality mesh, regardless of the order in which the 'bad' elements are processed. The algorithm can be parallelized by operating concurrently on non-overlapping and non-adjacent cavities. Furthermore, the per-cavity refinement process is completely independent. However, it is very difficult to identify non-conflicting cavities at compile-time due to input-dependent nature of the program. In addition, conflicts may arise at runtime due to cavity expansion during the refinement process. The conflict issue, however, makes this application a suitable candidate for exploiting the conflict-detection and rollback properties of the transactional memory. The program delaunay is a reimplementation of Kulkarni et al.'s earlier work [13].

In a transactional-memory implementation of the Delaunay algorithm, multiple threads choose their elements from a work-queue and refine the cavities as separate transactions. The conflict detection capabilities provided by the transactional-memory system detect any inter-cavity conflicts at runtime and the conflicting transactions are then rolled back.

Table 1 shows the dynamic characteristics of tasks corresponding to the critical sections in the program. Almost all the execution is spent in critical sections, most significantly delaunay.c:237, which computes the cavity refinement. Note that although the critical sections at lines 218 and 262 have high dependence densities (the former one coordinates concurrent accesses to a shared worklist, and the latter updates a shared bookkeeping variable), their coverage is very low. This led to the classification of 'low dependence density' assuming tens of threads. The implementation of delaunay randomizes the order of tasks in the initial worklist list; this corresponds to our assumption of an un-informed scheduler.

#### 4.1.2 genome

genome implements a gene sequencing program that reconstructs the most likely original gene sequence from a large pool of unmatched gene segments. The basic data structure is a hash table for storing unique unmatched gene segments. In a parallel scenario, each thread tries to add to its partition of currently matched segments by searching the shared pool of unmatched segments. Since multiples threads try to access the same segment, these accesses are executed as transactions within each thread.

Table 2 details the five critical sections in this application. The first and the third critical sections (lines 288 and 386) have the maximum coverage and hence their dependence densities are most critical to the parallel program performance. All of these critical sections protect accesses to the shared data structures such as the hash table and the shared segment pool. Access to these data structures are mostly independent and hence an un-informed scheduler can easily achieve high degrees of parallelism. The critical section at line 360, does not have any conflicts, i.e., both dependence density values are 0. Nevertheless, the critical section is necessary as conflicts may occur with different program inputs; this critical section is short and does not significantly affect application performance. The important critical sections at lines 288 and 386 exhibit low dependence densities, hence the classification of this application.

#### 4.1.3 SSCA2

We focus the evaluation of the SSCA2 [1] benchmark on kernel 4, which computes the "betweenness centrality" metric for each node in a randomly generated graph; graph generation is done in another kernel of this benchmark. Two variants of the algorithm are implemented that execute in two different phases. The critical section at line 131 falls into one phase while the critical sections lines 306, 380 are executed in the other phase. For both variants of the kernel, a significant fraction of the execution is spent outside critical sections, hence certainly not contributing to task-interdependence. The dependence densities of the tasks are very low, indicating that even in the sections of the phase with potential inter-dependencies, the probability of a dependence violation due to concurrent task execution is very low.

#### 4.1.4 vacation

vacation implements a travel reservation system backed by a non-distributed, in-memory database. The nature of the application is similar to the SPECjbb2000 benchmark. The database consists of four tables: cars, rooms, flights, and customers. The first three have relations with fields representing a unique ID number, reserved quantity, total available quantity, and price. The table of customers tracks the reservations made by each customer and the total price of the reservations they made. The database tables are implemented as red-black trees. The workload consists of several client threads interacting with the database via the system's transaction manager. Each client thread can invoke one of the three main tasks: (1) make a reservation: The client checks the price of  $n$  items and reserves a few of them. (2) delete customer, once the total cost of a customer's reservation is computed and the customer is removed from the system, (3) update to item tables: For a reservation, add items to one of the tables (e.g., car, room, and flight) using the unique ID as the key. Items may also be removed, i.e., for a reservation, remove items from one of the tables using the unique ID as the key. Each task operates on one or more tables and operates within a transactional context to maintain atomicity, consistency, and isolation.

The vacation application spends most of its execution time inside critical sections that protect the execution of individual user tasks. All tasks have low dependence densities. Although the dependence density for the high contention input is slightly higher than for the low contention input, the classification of this program is the same for both input sets.

## 4.2 Medium dependence density

The tasks in the following two application have medium dependence density under the assumption that the execution environment provides few tens of parallel threads.

critical section [file:line number]	coverage [% of phase]	count	size [# insts]	read-set		write-set		dep-density	
				avg	total	avg	total	same	all
delaunay.c:218	0.12	24227	34.99	2.31	31749	1.0	2	0.9992	0.0038
delaunay.c:237	97.67	39470	17808	80.9	1665645	95.4	3601920	0.0003	0.0003
delaunay.c:248	0.30	15244	143	5.1	39102	2.44	22076	0.5668	0.0024
delaunay.c:262	< 0.01	1	13.0	4.0	4	2.0	2	1.0	< 0.0001

**Table 1.** Characterization of delaunay.

critical section [file:line number]	coverage [% of phase]	count	size [# insts]	read-set		write-set		dep-density	
				avg	total	avg	total	same	all
sequencer.c:288	73.25	4096	45882.6	801.6	1229771	66.9	272848	0.0115	0.0089
sequencer.c:360	0.03	4065	18.0	1.0	4065	1.0	4065	0	0
sequencer.c:386	22.76	126015	463.5	52.9	315939	3.0	315844	0.0040	0.0008
sequencer.c:399	0.21	4065	132.2	8.01	10708	3.0	10704	0.0003	< 0.0001
sequencer.c:474	0.51	7438	177.36	13.56	50816	3.8	22359	0.0007	< 0.0001

**Table 2.** Characterization of genome.

critical section [file:line number]	coverage [% of phase]	count	size [# insts]	read-set		write-set		dep-density	
				avg	total	avg	total	same	all
betweennessCentrality.c:131	2.14	46674	10	2.0	434	1.0	217	0.0046	0.0046
betweennessCentrality.c:306	30.80	376962	28	3.1	3604	1.1	3570	0.0053	0.0050
betweennessCentrality.c:380	5.28	107419	17	4.0	464	1.0	219	0.0321	0.0107

**Table 3.** Characterization of ssca2.

critical section [file:line number]	coverage [% of phase]	count	size [# insts]	read-set		write-set		dep-density	
				avg	total	avg	total	same	all
<i>low contention</i>									
client.c:170	90.82	52439	2464	125.7	946682	25.2	1070875	0.0012	0.0012
client.c:229	2.40	6566	515	65.6	143386	3.50	22328	0.0003	0.0006
client.c:239	4.95	6531	1079	92.3	198855	8.71	55326	0.0007	0.0004
<i>high contention</i>									
client.c:170	86.73	52330	3413	178.3	509495	18.8	661639	0.0023	0.0026
client.c:229	6.66	6603	2072	250.7	391928	18.2	55243	0.0067	0.0043
client.c:239	5.34	6603	1668	134.3	117524	11.6	51481	0.0023	0.0026

**Table 4.** Characterization of vacation.

#### 4.2.1 kmeans

kmeans implements a commonly used clustering technique in data mining applications. It clusters a set of objects based on their attributes into  $k$  partitions, where  $k$  is a user-specified parameter. It uses an iterative refinement heuristic that partitions the input data into an initial partition of  $k$  sets. It then calculates the mean point or centroid of the sets using a similarity function. For example, for spatial clustering, usually the Euclid distance is used to measure the closeness of two objects. The algorithm then constructs a new partition by associating each object with the closest centroid. The centroids are recalculated for the new clusters, and algorithm repeated by alternate application of these two steps until convergence, which is obtained when the objects no longer switch clusters (or alternatively centroids are no longer changed). In the parallel implementation, each thread works on one partition and contention occurs when multiple threads try to associate the same object to multiple partitions.

Table 5 shows the characteristics of the two critical sections in kmeans: one at line 164, for computing the cluster centroid, and one at line 178, for updating the shared task queue. The overall coverage of critical sections reported in Table 5 is significantly lower than the number reported in [20] since we removed an unnecessary critical section (normal.c:144) from this program. Both critical sections have low coverage and do not affect the overall execution performance. Although the second critical section exhibit conflicts,

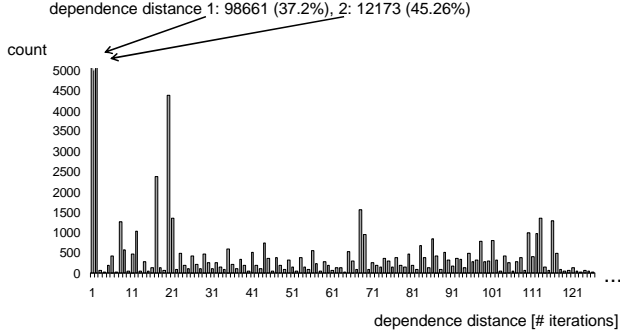
its overall impact is significantly lower than the first critical section (which has much lower dependence density). As expected, the dependence density for the low contention configuration is lower than the density of the high contention configuration. The available parallelism in the low contention case is just above thirty, hence the classification of this program in category “medium dependence density”.

#### 4.2.2 umt2k

umt2k is an unstructured mesh transport code [30]. This code is parallelized with a combination of MPI/OpenMP. For our study we focused on a sequential kernel in the sweep routine. In particular we studied the parallelization of a loop (snswp3d.c:357) the implements the sweep and covers about 50% of the overall sequential execution time of this benchmark. The execution of the loop forms a single phase wherein individual iterations of the loop execute as tasks. We re-factored the implementation of the loop body such that dependencies due to automatic variables access and reduction operations are avoided. Yet, some residual flow data dependencies among tasks remain. A scheduler must meet those dependencies, which reflect the energy transport that is simulated by this code, in the algorithmic order specified by the mesh topology. This order is featured indirectly as the original (sequential) iteration order of the loop.

critical section [file:line number]	coverage [% of phase]	count	size [# insts]	read-set		write-set		dep-density	
				avg	total	avg	total	same	all
<i>low contention</i>									
normal.c:164	1.81	350000	90.0	28.0	650600	13.0	520	0.0254	0.0242
normal.c:178	0.09	116662	13.0	2.0	2	1.0	1	0.9997	0.0455
<i>high contention</i>									
normal.c:164	3.50	250000	90.0	28.0	650300	13.0	260	0.0522	0.0497
normal.c:178	0.17	83330	13.0	2.0	2	1.0	1	0.9997	0.0464

**Table 5.** Characterization of kmeans.



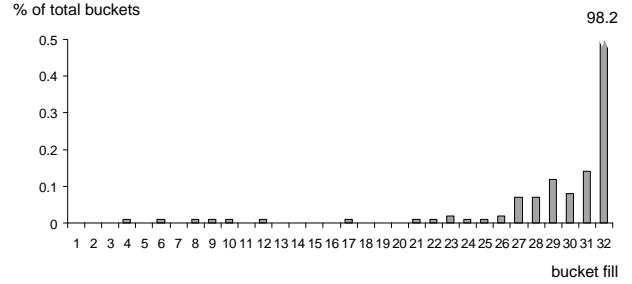
**Figure 2.** Dependence distance of iterations in the original iteration order.

Figure 2 shows a histogram of the distance of flow dependencies among iterations, where distance is measured as the number of iterations in the sequential schedule. The vast majority of tasks have a flow dependence on their predecessor or pre-predecessor; notice that the y-axis is shortened significantly in the graph to facilitate the presentation. Hence an un-informed scheduler that issues tasks in the sequence of the sequential schedule will achieve little parallelism.

The existence of occasional long-range dependencies, i.e., dependencies across tens or hundreds of iterations, suggest that a departure from the original iteration schedule could expose more parallelism. The iteration space of the loop can be understood as a directed acyclic graph where edges correspond to iterations (tasks) and edges correspond to data flow dependencies. The graph defines a partial order among tasks and any topological sort of the graph meets the flow dependencies. We computed from this abstract model of the iteration space a schedule that inserts independent tasks into buckets of fixed width  $k$ . If data dependencies are tight, then not all slots in a bucket will be filled with a task. A run time, all tasks within the same bucket can execute concurrently, while tasks in different buckets are executed in sequence. This methodology is similar to an inspector-executor model, where an execution schedule (in the original work [27] a communication schedule) is computed at run time based on dynamic data dependence information.

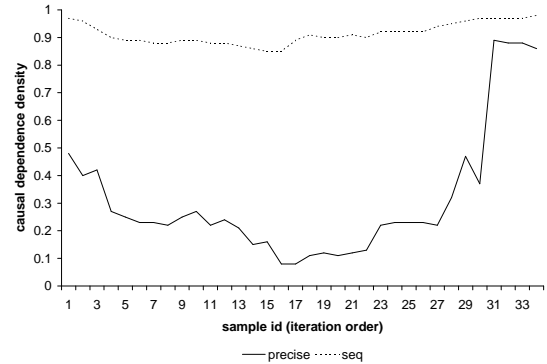
The results of this scheduling experiment are illustrated in Figure 3 for buckets of width  $k = 32$ . The histogram shows the fill of buckets in the final schedule: nearly all buckets can be filled with independent tasks, hence such algorithmic scheduler can clearly surpass the available parallelism exposed in the original schedule; again, the y-axis is shortened significantly in the figure to facilitate the presentation.

Table 6 specifies the characteristics of the tasks in umt2k. The dependence density is specified as a range and average, specifies the lowest and highest values observed in the sequence of samples. The density computed according the *causal\_chain\_seq* is close to 1, which reflects that an un-informed scheduler that adheres to the



**Figure 3.** Resorted iteration order: Fill of buckets of width 32 with iterations that are causally independent.

sequential execution order will have little success find parallelism. The density computed according to *causal\_chain\_precise* varies widely among samples (see Figure 4); the low values reflects the ability of an algorithmic scheduler to find parallelism within a window of 250 tasks (sample size). We observed that the average density dropped slightly as we increased the window size. Also, high density values were observed in early stages of the iteration space, whereas dependence densities dropped form samples in the middle of the iterations space. This reflects observation reflects the extent of the mesh and unfolding parallelism as the 'frontline' of the sweep through the mesh widens.



**Figure 4.** Causal dependence densities for different samples in the iteration space of umt2k.

### 4.3 High dependence density

#### 4.3.1 MySQL keycache

The keycache is a data structure in MySQL's MyISAM storage manager [22] that provides a cache of index blocks from database tables that reside in files on disk. Nearly every database operation (insert, query, update) accesses this structure. For our experiments, we used MySQL version 5.2 and the ATIS SQL benchmark in the distribution to synthesize a workload (insert followed by query)

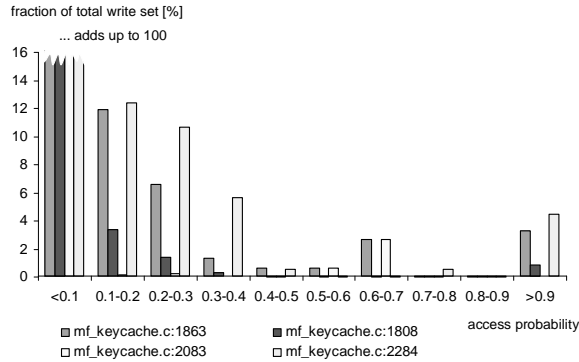
speculative region [file:line number]	coverage [% of phase]	count	size [# insts]	read-set		write-set		dep-density	
				avg	total	avg	total	seq	precise
snswp3d.c:357	100.0	9999	218.71	54.4	136804	38.5	13757	0.88-0.97 (0.31)	0.08-0.86 (0.91)

**Table 6.** Characterization of umt2k.

on an artificial database. The execution of the ATIS benchmark constitutes a single phase.

Several critical sections guard the shared data structures in the keycache and a non-negligible fraction of the execution time is spent in two critical sections at `mf_keycache.c:1808,1863`. The dependence distance in those critical sections is close to one, hence almost any pair of concurrent tasks will encounter a conflict. This suggests that the choice to implement concurrency control with a single lock, as done in the implementation that we studied, is reasonable.

Figure 5 illustrates the write access behavior of the critical section with the highest coverage in execution time. Bars of a certain color add up to 100 units on the y-axis (the axis is shortened for presentation purpose), reflecting the fraction of the locations written by tasks corresponding to a certain critical section. Each location falls into some bin on the x-axis. The x-axis plots the probability that some task accesses a variable in the bin. The graph illustrates that a large majority of variables are written occasionally, i.e., with probability  $< 0.1$  by any one critical section. However, some variables, the probability of write access by a randomly chosen task is very high ( $> 0.9$ ). This observation conveys that the high dependence density in this workload can be attributed to very few variables.



**Figure 5.** Likelihood of write access of individual variables in the four most prominent critical sections of the keycache workload.

A closer study of the in source code unveils that most if not all of the ‘highly contended’ locations correspond to counters serving statistics or guiding the replacement policy of the cache. Note that such counters would be a definite obstacle to a straightforward application of transactional memory; techniques have been proposed to overcome the frequent task rollbacks in such case [21, 9].

#### 4.4 Effect of varying the dependence granularity

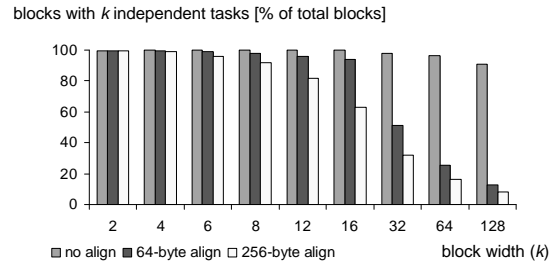
So far, the task profiling tool recorded read and write sets as word-aligned addresses (Section 3), i.e., an inter-task dependence is detected if different tasks read and write the same word. Systems that support the detection of interference at run time commonly choose to track addresses at cache-line granularity (32 bytes to 128 bytes is common), which can lead to an increased number of reports of inter-task dependencies. Some of such reports can be spurious, i.e., they reflect access to the same cache line but accesses may not actually interfere if they target different offsets in the line.

An increased number of interference reports also increases the dependence density and hence affects the available parallelism that we determine for an application. To assess the effects of coarser grain dependence detection, we configure the task profiler to track addresses at 64 byte alignment; also we consider writes by different tasks to the same line as task inter-dependence (although no communication is facilitated through those writes). Table 8 reports the dependence densities for selected critical sections.

We observe that dependence densities can increase significantly due to false sharing. For example in the case of vacation, certain critical sections have medium dependence density (assuming tens of threads) such that scheduling becomes a relevant performance factor.

For umt2k, we illustrate the effects of false sharing by studying the occurrence of dependencies using the algorithmic scheduling model described in Section 4.2.2. The scheduler arranges independent tasks in buckets of size  $k$  or smaller under the assumption of different address alignments.

Figure 6 shows the percentage (y-axis) of  $k$ -size buckets (x-axis) where all  $k$  slots could be filled with independent tasks. Generally, buckets of larger sizes do not fill up as well as smaller buckets. In the case where addresses are tracked precisely (no align), for example, 98.2% of buckets of size  $k = 32$  can be filled completely, while just above 90% of the buckets can be filled when  $k = 128$ . The second observation is that the increase of address alignment significantly reduces the ability of the scheduler to fill up the buckets with independent tasks.



**Figure 6.** Algorithmic scheduling for umt2k: Fill of buckets of various width with no, 64-byte, and 256-byte address alignment.

False sharing in umt2k does not only affect inter-task dependencies but also puts a significant penalty on the shared memory access latency. We executed the sweep phase of the benchmark on a Power5 multiprocessor, scheduling the execution of tasks (individual loop iterations) to processor cores in a round robin manner. Processor cores executed the tasks in order, irrespective inter-task dependencies (i.e., concurrency control was omitted for the purpose of this limit study, hence the result of the computation could have been incorrect). Execution on 8 cores achieved a speedup of 3 over single thread performance. This less than ideal speedup can be attributed entirely to increased latencies in the memory access path due to the sharing of lines in caches associated with different cores and processors.

critical section [file:line number]	coverage [% of phase]	count	size [# insts]	read-set		write-set		dep-density	
				avg	total	avg	total	same	all
mf_keycache.c:1808	3.28	277134	271.6	28.2	481	47.2	1151	0.9996	0.2577
mf_keycache.c:1863	3.46	277134	286.2	23.7	253	41.6	699	0.9996	0.9946
mf_keycache.c:2083	0.39	13278	673.5	306.0	9514	330.4	4397	0.9992	0.9946
mf_keycache.c:2271	< 0.01	9768	16.0	5.0	5	4.0	4	0.9990	0.9945
mf_keycache.c:2284	0.23	13278	396.1	42.4	198	60.2	193	0.9992	0.2550
mf_keycache.c:2552	< 0.01	11	124.0	10.0	30	16.0	61	0.9990	0.9998

**Table 7.** Characterization of MySQL keycache.

critical section [file:line number]	dep-density	
	same	all
del aunay		
del aunay.c:237	0.0005	0.0005
genome		
sequencer.c:288	0.0509	0.0506
kmeans (low contention)		
normal.c:164	0.0886	0.0845
vacation (high contention)		
client.c:170	0.0334	0.0323
client.c:229	0.0110	0.0180
client.c:239	0.0271	0.0186

**Table 8.** Dependence densities with dependence checking at 64-byte alignment.

## 5. Related work

Saltz et al. first proposed [27] using the inspector-executor principle for parallelizing applications with irregular memory access patterns. In this approach, the inspector phase determines the dependencies between units of work and the executor phase performs the computation in parallel. This approach is suitable for applications where the execution schedule can remain stable at runtime and hence the cost of the inspector phase can be amortized. This approach is not suitable for applications where execution schedules have to adapt dynamically, e.g., the Delaunay mesh generation [13]. Recently, Kulkarni and Pingali have demonstrated that *optimistic parallelism* is an effective tool for parallelizing such applications [15].

The Transactional Memory (TM) programming model has been proposed for managing concurrency in irregular problems. TM [11] was first introduced by Herlihy and Moss as an alternative to concurrent programming using locks. Since then, the TM approach has been investigated in software, hardware and in hybrid schemes that combine software and hardware implementations [28, 6, 8, 20, 2]. While there has been significant work in developing TM infrastructure, behavior of TM-enabled applications has not been analyzed in depth, with some notable exceptions [15, 10]. To the best of our knowledge, no prior work has abstracted program dependence via quantitative means and used it to expose potential parallelism.

Thread level speculation (TLS) [31, 12, 7, 29] has been introduced as a technique to speedup serial applications that are notoriously hard to parallelize, such as SPECint. In this approach, tasks are identified in the application by a compiler or the user, and the system is responsible for detecting dependence violations and restarting the offending tasks. Tasks are typically selected as either loop iterations or function continuations. Compiler instrumentation and profiling has been used to select profitable tasks [17, 33]. In addition, several studies have looked at the ideal parallelism that can be obtained with TLS systems [16, 25, 19].

Runtime program profiling has been used extensively to support optimizations like instruction scheduling for super-scalar processors, program hot-spot analysis, and speculative execution [5, 24]. Data dependence profiling has also been used in the context of TM and TLS, again, to identify and select the most profitable tasks. Ex-

amples include TEST [3], JRPM [4] and the IPOT TaskFinder [32]. In this work the authors use models based on data dependence to analyze and rank tasks with respect to profitability for execution. However, each task is considered in isolation and there is no scheduler that selects tasks in order to maximize parallelism.

## 6. Concluding remarks

In this paper, we presented a quantitative model that captures data and causal dependencies in parallel programs. We applied this model to determine parallelization opportunities in programs for which the parallelization potential cannot be analyzed statically (e.g., those with irregular memory accesses). Our model abstracts runtime aspects of the execution and represents dependencies as a density computed over the number of executed instructions. This density metric can capture data dependencies in both ordered and unordered critical sections. We have implemented this model in a dynamic binary instrumentation and profiling tool and applied this profiler to several parallel applications requiring ordered and unordered execution of critical sections.

Given the amount of available parallelism, the dependence density metric can be used to predict potential concurrency and scalability properties of parallel applications. Applications fall into three categories: low, medium and high dependence density. Applications with low dependence density are naturally good candidates for optimistic concurrency, while application with medium dependence density may require a scheduler that is aware of the algorithmic dependencies for optimistic concurrency to be effective. Applications with high dependence density may not be amenable to parallelization.

## Acknowledgments

We would like to acknowledge members of the transactional memory project at IBM Research for providing the basis for this work. We would like to thank the STAMP team at Stanford for giving us early access to their benchmark suite. We would also like to thank Keshav Pingali for providing access to the Delaunay mesh generation application. We also thank the reviewers for their thorough reading and insightful comments.

## References

- [1] D. Bader and K. Madduri. Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. In *Proc. of HiPC 2005*, pages 465–476, December 2005.
- [2] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. In *Proc. of ISCA'07*, pages 81–91, 2007.
- [3] M. K. Chen and K. Olukotun. TEST: A Tracer for Extracting Speculative Thread. In *Proc. of CGO'03*, pages 301–314, 2003.
- [4] M. K. Chen and K. Olukotun. The Jrpm System for Dynamically Parallelizing Java Programs. In *Proc. of ISCA'03*, pages 434–445, 2003.
- [5] T. Chen, J. Lin, X. Dai, W.-C. Hsu, and P.-C. Yew. Data Dependence Profiling for Speculative Optimizations. In *Proc. of Compiler Construction'04*, pages 57–72, 2004.
- [6] D. Dice and N. Shavit. Understanding Tradeoffs in Software Transactional Memory. In *Proc. of CGO'07*, 2007.
- [7] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proc. of ASPLOS'98*, 1998.
- [8] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proc. of OOPSLA'03*, pages 14–25, 2006.
- [9] T. Harris and S. Stipic. Abstract Nested Transactions. In *Proc. of TRANSACT'07*, 2007.
- [10] M. Herlihy and E. Koskinen. Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects. Technical Report CS-07-08, Department of Computer Science, Brown University, July 2007.
- [11] M. Herlihy and J. E. B. Moss. Transactional Memory: Architecture Support for Lock-free Data Structures. In *Proc. of ISCA'93*, pages 289–300, 1993.
- [12] V. Krishnan and J. Torrellas. Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor. In *Proc. of ICS'98*, 1998.
- [13] M. Kulkarni, L. P. Chew, and K. Pingali. Using Transactions in Delaunay Mesh Generation. In *Workshop on Transactional Memory Workloads (WTW'06)*, 2006.
- [14] M. Kulkarni and K. Pingali. Scheduling Issues in Optimistic Parallelization. In *Proc. of IPDPS'07*, pages 1–7, 2007.
- [15] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic Parallelism Requires Abstractions. In *Proc. of PLDI'07*, pages 211–222, 2007.
- [16] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proc. of ISCA '92*, pages 46–57, 1992.
- [17] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits program structure. In *Proc. of PPOPP '06*, pages 158–167, 2006.
- [18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of PLDI'05*, 2005.
- [19] P. Marcuello and A. Gonzalez. A Quantitative Assessment of TLS Techniques. In *Proc. of IPDPS'00*, 2000.
- [20] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proc. of ISCA'07*, pages 69–80, 2007.
- [21] E. J. B. Moss and T. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. In *In Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages*, Oct 2005.
- [22] MySQL - The World's Most Popular Open Source Database. <http://www.mysql.com>.
- [23] R. Narayanan, B. Ozisikyilmaz, J. Zamberno, G. Memik, and A. Choudhary. MineBench: A Benchmark Suite for Data Mining Workloads. In *Proc. of IISWC'06*, pages 182–188, 2006.
- [24] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware Atomicity for Reliable Software Speculation. In *Proc. of ISCA'07*, pages 174–185, 2007.
- [25] J. Oplinger, D. Heine, and M. Lam. In Search of Speculative Thread Level Parallelism. In *Proc. of PACT'99*, pages 303–313, 1999.
- [26] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *Proc. of PPOPP'06*, pages 187–197, March 2006.
- [27] J. Saltz, R. Mirchandaney, and K. Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5), 1991.
- [28] N. Shavit and D. Touitou. Software Transactional Memory. In *Proc. of PODC'95*, pages 204–213, 1995.
- [29] J. G. Steffan and T. C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proc. of HPCA'98*, 1998.
- [30] The UMT Benchmark Code. <http://www.llnl.gov/ascii/purple/-benchmarks/limited/umt>.
- [31] T. Vijaykumar and G. S. Sohi. Task Selection for a Multiscalar Processor. In *Proc. of MICRO'98*, 1998.
- [32] C. von Praun, L. Ceze, and C. Cascaval. Implicit Parallelism with Ordered Transactions. In *Proc. of PPOPP'07*, pages 79–89, 2007.
- [33] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler Optimization of Memory-Resident Value Communication Between Speculative Threads. In *Proc. of CGO'04*, pages 39–52, 2004.