TIGHTFIT: Adaptive Parallelization with Foresight

Omer Tripp*
Tel-Aviv University
omertrip@post.tau.ac.il

Noam Rinetzky†
Tel-Aviv University
maon@cs.tau.ac.il

ABSTRACT

Irregular applications often exhibit data-dependent parallelism: Different *inputs*, and sometimes also different *execution phases*, enable different levels of parallelism. These changes in available parallelism have motivated work on adaptive concurrency control mechanisms. Existing adaptation techniques mostly learn about available parallelism *indirectly*, through runtime monitors that detect pathologies (*e.g.* excessive retries in speculation or high lock contention in mutual exclusion).

We present a novel approach to adaptive parallelization, whereby the effective level of parallelism is predicted *directly* based on input features, rather than through circumstantial indicators over the execution environment (such as retry rate). This enables adaptation with *foresight*, based on the input data and not the run prefix. For this, the user specifies input features, which our system then correlates with the amount of available parallelism through offline learning. The resulting prediction rule serves in deployment runs to foresee the available parallelism for a given workload and tune the parallelization system accordingly.

We have implemented our approach in TIGHTFIT, a general framework for input-centric offline adaptation. Our experimental evaluation of TIGHTFIT over two adaptive runtime systems and eight benchmarks provides positive evidence regarding TIGHTFIT's efficacy and accuracy.

Categories and Subject Descriptors

D.1.3 [**Software Engineering**]: Concurrent Programming

General Terms

Algorithms, Performance, Experimentation, Measurement

*Supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement n° [321174-VSSC]

†Supported by the EU project ADVENT, grant number: 308830

Keywords

offline learning, irregular applications, data-dependent parallelism, adaptive software parallelization, STM

1. INTRODUCTION

This paper addresses the problem of parallelizing irregular applications, whose available parallelism is data dependent. Contrary to regular applications (*e.g.* scientific programs that manipulate dense arrays), where dependencies between computations are identical for all inputs, an irregular application has a different dependence structure per different data inputs, which leads to fluctuating parallelism across inputs, and sometimes also execution phases. As an illustrative example, minimum spanning tree (MST) algorithms typically have more parallelism over dense graphs, where it is less likely for tasks to work on common nodes or edges.

Irregular applications are widespread [13]. Common examples include graph algorithms, such as Boruvka's and Kruskal's MST algorithms and Dijkstra's single-source shortest path algorithm; scientific applications like the Barnes-Hut and Discrete Event Simulation algorithms; machine-learning and data-mining algorithms, such as Agglomerative Clustering and Survey Propagation; and applications in the area of computational geometry, including Delaunay's mesh-refinement and triangulation algorithms.

Effective parallelization of irregular applications is challenging because the available parallelism is input sensitive. Fixing a single synchronization scheme, treating all inputs uniformly, might either (i) exploit the available parallelism poorly for high-parallelism inputs if synchronization is conservative (*e.g.* coarse-grained locking), or (ii) yield high overheads for low-parallelism inputs if synchronization is permissive (*e.g.* many retries in speculative execution of conflicting transactions).

Adaptive concurrency control. A natural means of addressing data-dependent parallelism is adaptation, whereby the parallelization system changes its behavior in response to changes in available parallelism. Recent studies have proposed a variety of techniques for identifying such changes, and more specifically, estimating available parallelism on the fly. Examples include monitoring abort/commit ratio in software transactional memory (STM) [5, 40], or tracking access patterns to shared data structures at the early stages of execution [8]. (See Section 7 for a comprehensive discussion.)

These as well as other existing techniques estimate the parallelism enabled by the input workload *indirectly*, based on properties of the execution environment rather than prop-

erties of the input. The motivation is clear: Deriving useful input characteristics require semantics understanding of the application at hand, whereas profiling system behaviors (like abort/commit ratio or lock contention) provides a general and tractable basis for automatic adaptation. The disadvantage, however, is that adaptation is guided by *hindsight* judgments over the behavior of the execution environment during the run prefix. This can potentially lead to poor performance at the beginning of the run, as well as misguided adaptation if parallelism changes across phases (*i.e.* when the future is not predictable from the past).

Ideally, the runtime system would make adaptation decisions *directly*, with *foresight*: Given an input workload, the system would know in advance, based on inspection of the input itself, how much parallelism that workload permits, and tune its behavior accordingly [17, 12]. Designing foresight-guided adaptation mechanisms is challenging: It requires uncovering the relationship between inputs and available parallelism at runtime, and with low overhead [32].

Our approach. We present a novel approach for enabling foresight-based adaptation. The main idea is to utilize offline analysis—and more specifically, a heavyweight learning algorithm—to characterize the parallelism permitted by different inputs. The runtime system then makes use of the artifacts from offline analysis to derive adaptation decisions from the current state of the workload at hand.

What complicates learning is that the input data is often a complex data structure, like a graph. To address this challenge, we ask the user to provide a *feature extraction function*, mapping the concrete input to a set of features that are amenable to learning (*e.g.* the number of nodes in the graph, the number of edges, the graph's density, etc). We require this specification because feature extraction is a challenging problem for an automated algorithm. It is our impression and experience that providing the specification places low burden on a user intimate with the target application, as the entailed reasoning is relatively simple. Furthermore, the specification is concise and, most importantly, the offline learning system is robust to user errors. (See Section 2 for further discussion.)

Based on the specification of input features, our approach decomposes into the following three phases:

1. ***Per-system learning.*** For the given adaptive parallelization system, converge (once per all applications) on the most favorable execution mode of the system per different profiles of available parallelism.
2. ***Offline learning.*** For every application, build a prediction function from input features to available parallelism using offline learning over representative workloads.
3. ***Runtime parallelization.*** At runtime, the system periodically calls the user-provided feature extraction function to obtain the features of the current data set. It then performs a lookup to find which mode of the parallelization system best matches these features, and sets the system to that mode.

For the second phase of estimating available parallelism, which is the main focus of this paper, we analyze data dependencies. A data dependency arises between two statements during (sequential) execution if both access a common memory location, and at least one of them writes it. Intuitively, this is an indication that the two statements might interfere with each other during concurrent execution.

Traditionally, compile-time parallelization transformations have been governed by *qualitative* analysis of data dependencies, requiring absence of data dependencies between code blocks as the precondition for their parallel execution. A fundamental observation of this paper is that dynamic data dependencies expose fine-grained information about the available parallelism. This includes not only the volume of data dependencies between tasks that can potentially run in parallel (*i.e.* the density of the dependence graph), but also the structure of such dependencies, which discloses *e.g.* whether two tasks have cyclic dependencies. This provides a principled basis for offline estimation of available parallelism that abstracts away low-level deployment details (such as the size of the cache, the number of cores, the number of executing threads, etc).

An important emphasis of our approach is on effective user interaction: The application designer, who is best aware of pertinent workload features, communicates this information through a concise and lightweight specification. The adaptation algorithm, in turn, leverages this information to build a bridge—using statistical analysis—between workload characteristics and available parallelism, which is known as a challenging task for fully automated systems [32].

Scope and limitations. We expect the application designer to provide candidate features as well as representative workloads for profiling. We assume that the program is already parallel, or at least contains atomicity annotations, and so the notion of atomic tasks is specified over the program's code. We further assume that the criterion for correct parallel execution is serializable execution of atomic blocks (or transactions), which TIGHTFIT guarantees.

A dependent, and more subtle, assumption is that parallel tasks communicate only by modifying the shared memory, as is the case *e.g.* with applications using STM, and thus data dependencies are a reliable model of runtime conflicts.

A main source of complexity in TIGHTFIT is the learning process. This process entails offline analysis, as well as user involvement, but in return it enables low-overhead input-centric adaptation. At present, our prototype has limited inference capabilities, and so the user has to choose which statistical learning algorithm to apply (though this, again, is a question of performance/accuracy and not correctness). We intend to reduce this configuration burden in the future by introducing inference capabilities into TIGHTFIT.

Contributions. The principal contributions of this paper are the following:

- ***Adaptation with foresight.*** We present a novel solution for the problem of adaptive parallelization, focusing on the direct connection between input features and parallelism. This is achieved via expensive offline analysis of training runs (backed by user-provided input features), alongside low-overhead, input-centric runtime adaptation.
- ***Adaptation based on input features.*** We make the relationship between the input data and available parallelism amenable to learning by abstracting the data as a set of features according to a user-provided specification. (We believe that this novel idea can be reused in other contexts.)
- ***Parallelism characterization via analysis of dependencies.*** We derive *quantitative* as well as *structural* characteristics of data dependencies to estimate the available parallelism, thereby abstracting away

```
Graph g = /* read input graph */;
Graph mst = g.getNodes();
List worklist = g.getNodes();
@atomic foreach (Node nd in worklist) {
  Node nbr = minWeight(nd, g.getAdjacent(nd));
  Node nnd = edgeContract(nd, nbr);
  mst.addEdge(nbr, nnd);
  worklist.add(nnd); }
```

Figure 2: The Boruvka MST algorithm

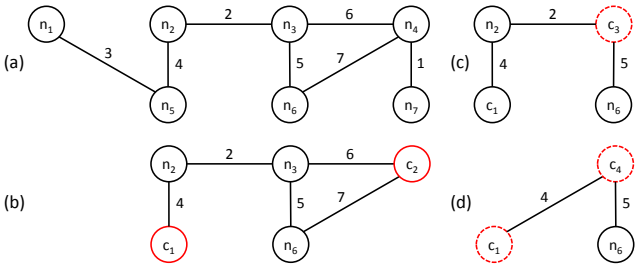

Figure 3: Different revisions of an input graph under the Boruvka MST algorithm

implementation-specific details. This goes beyond the standard approach of treating data dependencies *qualitatively* as a clear-cutting criterion for disjoint parallelism.

- ***Implementation and evaluation.*** We have implemented our approach in TIGHTFIT. We describe our experiments with TIGHTFIT over two different adaptive parallelization systems and a suite of eight benchmarks. The results provide solid evidence in support of our approach. (TIGHTFIT is available online at [3].)

2. OVERVIEW

In this section, we motivate our approach with reference to Boruvka's MST algorithm. We then walk the reader through Figure 1, which summarizes the entire flow of the TIGHTFIT system.

2.1 Boruvka's algorithm

Figure 2 shows Boruvka's algorithm. The algorithm computes a minimum spanning tree by reducing the input graph to a single node through successive application of edge contraction. In each iteration, a graph node `nd` is selected nondeterministically, a minimum-weight neighbor `nnd` of `nd` is obtained, and the edge between `nd` and `nnd` undergoes contraction (becoming part of the MST).

For a sparse input graph, this permits a high level of parallel work at the beginning, where different contractions are applied to disjoint regions of the graph, but ultimately, as the graph grows smaller, the available parallelism gradually decays [21]. These two cases are illustrated in Figure 3: The transition from state (a) of the graph to state (b) is via two parallel, non-overlapping edge contractions: $(\mathtt{n}_1, \mathtt{n}_5) \rightarrow \mathtt{c}_1$ and $(\mathtt{n}_4, \mathtt{n}_7) \rightarrow \mathtt{c}_2$. However, the application of contraction $(\mathtt{c}_2, \mathtt{n}_3) \rightarrow \mathtt{c}_3$, shown as state (c), cannot run in parallel with its succeeding contraction, $(\mathtt{n}_2, \mathtt{c}_3) \rightarrow \mathtt{c}_4$, which appears in (d). Both access a common region in the graph.

```
features Graph:g {
  "nnodes":  { g.nnodes(); }
  "density": { (2.0 * g.nedges()) /
                          g.nnodes() * (g.nedges()-1); }
  "avgdeg":  { (2.0 * g.nedges()) / g.nnodes(); } ....  }
```

Figure 4: Fragment of the TIGHTFIT specification for the `Graph` type (*cf.* Figure 2)

2.2 Flow

The TIGHTFIT system breaks the parallelization process into several phases, which we discuss in turn.

Parallelization modes. For a given adaptive parallelization system, TIGHTFIT needs to establish a mapping from parallelism levels to effective execution modes of the adaptive system. This is done once per each system. The parallelism-to-mode mapping, shown as the "Sys.-level para. $\mapsto$ mode" box in Figure 1, can either be specified by the user or it can be learned automatically.

In this paper, we place more emphasis on the problem of estimating application-specific per-input parallelism, and so to compute the parallelism-to-mode mapping, we resorted to the standard solution of running the system on a synthetic benchmark whose available parallelism is parametric to find effective threshold values for switching between parallelization modes [38]. (The underlying assumption is that different modes correspond to different parallelism levels.) We expand on this step in Section 4.

Offline adaptation. The main focus of this paper is on learning an input-centric application-specific adaptation rule. To make the learning setting induced by offline adaptation feasible (especially in the case of complex inputs, like a graph), we ask the user to "simplify" the input description by providing a feature extraction function that reduces the input to a vector of simple features. This is summarized as the double-framed "Input $\mapsto$ features" box in Figure 1. Figure 4 illustrates the specification format for the example of a `Graph` data structure.

Offline learning of adaptation rules. (**Input $\mapsto$ features**) For every application, we learn an input-centric adaptation rule. To make the learning procedure feasible, especially in the case of complex inputs, like a graph, we ask the user to "simplify" the input description by providing a *feature extraction function* that reduces the input to a vector of simple features. This is summarized as the double-framed "Input $\mapsto$ features" box in Figure 1. Figure 4 illustrates the specification format for the example of a `Graph` data structure.

The required specification puts little responsibility on the user's shoulders, in that even if the specification is incomplete or simply irrelevant, correctness (*i.e.* serializability) of parallel execution is guaranteed. The user can only affect the efficacy of adaptation decisions, at most degrading performance if providing a bad specification. Moreover, the specification is not application specific. All graph algorithms, for example, can share the specification in Figure 4.

TIGHTFIT favors a comprehensive specification, including features that may prove useless or irrelevant rather than excluding them. This is because TIGHTFIT's learning algorithm applies regression, seeking correlations between input features and (estimated) parallelism levels, which automatically prunes irrelevant (*i.e.* weakly correlated) features, as-

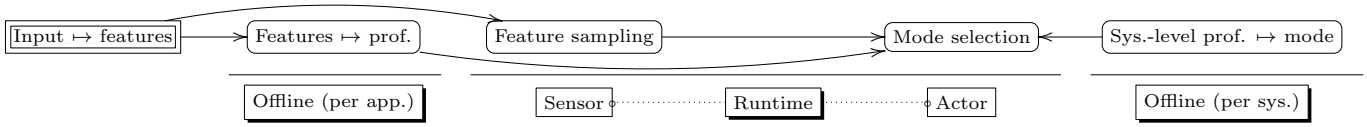

Figure 1: Outline of the complete TIGHTFIT flow

signing them low weight in the adaptation rule. This makes such features harmless, and thus the more exhaustive the specification is, the better the adaptation rule becomes.

Since feature extraction is done not only offline, but also online, during parallel runs, features should be efficiently computable. If at runtime feature computation requires too many cycles (*e.g.* counting how many cycles or simple paths a graph contains), then the performance benefits of adaptation are obviated. As an example, the cost of methods `nnodes` and `nedges` in Figure 4 should be low.

In addition, to ensure the statistical significance of the regression algorithm correlating input features with parallelism, we ask the user to characterize the application's input space by providing legal ranges for input parameters. TIGHTFIT features a built-in harness for sampling inputs at random based on the user's characterization of parameter ranges.

(**Features ↦ parallelism profile**) The next step in offline adaptation, given the availability of input features, is to estimate available parallelism over the training runs. Our main observation here is that profile-guided analysis of data dependencies between tasks designated for concurrent execution permits effective measurement of available parallelism. This is because at runtime, data dependencies translate into conflicting accesses to shared memory, which mandate synchronization, thereby limiting parallelism. Moreover, compared to more concrete measures of available parallelism (like direct measurement of running time over different inputs), data dependencies are less coupled with low-level deployment details (like the hardware architecture, number of threads, caching, etc), which makes the learned input-to-parallelism correlations more significant as well as robust to changes in the parallelization system and/or deployment configuration.

For the example of Boruvka, there are indeed no data dependencies between the first two loop iterations, but the next two iterations, performing overlapping contractions, generate data dependencies over c_3. That is, there is an increase in the number of data dependencies as the computation evolves, which is compatible with the observation that the available parallelism in Boruvka gradually decays.

Quantitative abstraction of data dependencies enables offline generation of input-centric parallelism profiles: The target application is run on different inputs, a dependence graph is computed for each trace, and the density and shape of the graph are analyzed to derive a general (rather than deployment-sensitive) measure of available parallelism. This exposes the relationship between input properties and parallelism level, allowing prediction of the available parallelism for new inputs, as summarized in the "Features ↦ prof." box in Figure 1. This analysis is the subject of Sections 3 and 4.1.

(**Parallelism profile ↦ mode**) The remaining task is to correlate between parallelism profiles with different execution modes of a particular adaptive parallelization system. This correlation is achieved via a synthetic benchmark suite that enables parametric control over the amount of parallelism in a run. This task is summarized in the "Sys.-level prof. ↦ mode" box in Figure 1, and is the subject of Section 4.2.

Runtime parallelization. The final stage in the TIGHTFIT flow is online parallelization. Here the offline adaptation stragey is utilized by sampling—at the beginning of the run, and possibly also throughout the run (when starting an atomic block) if there are phase transitions—the feature values for the input at hand. This is done, based on the same feature extraction function serving for offline analysis, by the "Sensor" module, which flows this information to the "Actor" component. The "Actor" module then makes a mode recommendation (the "Mode selection" box) based on the composition of the "Features ↦ prof." and "Sys.-level prof. ↦ mode" functions computed offline. In Section 5, we discuss two adaptive parallelization systems with which we evaluated TIGHTFIT.

3. PREDICTING AVAILABLE PARALLELISM FROM DATA DEPENDENCIES

In this section, we explain how the available parallelism in a given (sequential) execution trace is estimated by analysis of data dependencies.

3.1 Atomicity-aware dependence graphs

A sequential trace is a sequence $[\ldots(\sigma, s, \sigma')\ldots]$ of transitions, where s is a primitive statement and σ and σ' are the prestate and poststate, respectively. Abstracting a trace as a dependence graph is standard [35]. For the purpose of this paper, we define a slight variant, which we refer to as an *atomicity-aware dependence graph* (AADG). We assume that the program is annotated with `atomic` $\{\ldots\}$ sections, and that the semantics records entry and exit events corresponding to entering and leaving atomic sections.

This yields, for a given sequential run of the program, a partitioning of the transitions coming from atomic sections into distinct *atomic blocks*. (Note that different executions, or instances, of the same atomic section correspond to different atomic blocks.)

Trace τ is abstracted as an AADG as shown in Figure 5:

1. Every transition $t = (\sigma, s, \sigma')$ occurring within an atomic block b is mapped to a node (t, b).
2. For each memory location l read by t, we draw edges to all nodes (t', b'), such that block b' differs from b and l is written by t' (*i.e.* there is flow dependence between t and t', and these transitions occur in different atomic blocks).
3. For each location l written by t, we draw edges to all nodes (t', b'), such that block b' differs from b and l is either read or written by t' (*i.e.* there is either anti dependence or output dependence between t and t',

Offline Estimation of Parallelism by Analysis of Dependencies

Input:
τ: execution trace with atomicity annotations

OFFLINEESTIMATE: [returns estimated contention, cyclic dep.s]
let $G = \emptyset$ [initialize empty dependence graph]
for each transition t occurring in atomic block b in τ:
insert node (t, b) into G
for each mem. loc. $l \in r(t)$: [$r(.)$ denotes readset]
for each node $(t', b') \in G$ s.t. $l \in w(t')$, $b \neq b'$:
insert edge $(t', b') \leftarrow (t, b)$ into G
for each mem. loc. $l \in w(t)$: [$w(.)$ denotes writeset]
for each node $(t', b') \in G$ s.t. $l \in r(t') \cup w(t')$, $b \neq b'$:
insert edge $(t', b') \leftarrow (t, b)$ into G
return (density G, cdep G)

Figure 5: Offline alg. for parallelism estimation

which occur in different atomic blocks).
Note that the algorithm is parametric in the specification of readsets and writesets (r and w), as well as in the grain of trace transitions [35]. (We explain the algorithm's output soon.)

The above steps yield a standard dependence graph modulo two adjustments: First, nodes point to their enclosing atomic block, and second, there are no intra-block dependence edges. Intuitively, this enables *qualitative checking*, as well as *quantitative measurement*, of dependencies between statements belonging in distinct atomic blocks designated for parallel execution.

Note, importantly, that TIGHTFIT computes AADGs solely over sequential traces (and not parallel ones). However, assuming the parallelization system guarantees serializability, there is no loss of generality here. This is because a parallel trace can be serialized, and thus its AADG is the same as that of its corresponding sequential run, allowing offline analysis to consider only sequential executions.

3.2 Quantifying dependencies

Given an AADG G, we measure two aspects of the parallelism it permits:

Contention. Contention is interpreted as the level of (potential) interference between tasks designated for parallel execution. This corresponds naturally to the density of the AADG (when viewed as an undirected graph):

$$\mathsf{density}\ G = 2 \cdot |G.E| \,/ (|G.V| \cdot (|G.V| - 1)).$$

We obtain a normalized measure of the intensity (or proportion) of dependencies between atomic blocks. (Data dependencies between transitions in the same atomic block are suppressed to concentrate on inter-task dependencies.)

Cyclic dependence. Another measure of parallelism is the level to which tasks exhibit cyclic dependencies. This situation, illustrated in Figure 6, arises when distinct atomic blocks access two or more memory locations in opposing orders. In Figure 6, this is shown for four transitions from two atomic blocks. The upper-left and bottom-right transitions are dependent due to memory location l_1 (output dependence), whereas the other two transitions are dependent due to l_2 (flow dependence). Measuring cyclic dependencies—beyond contention—is useful for synchronization, because some protocols work well under high contention, but cope poorly with cyclic dependencies [25, 26].

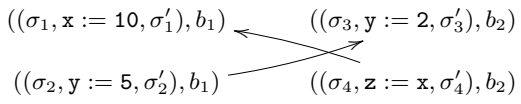

Figure 6: Illustration of a sitaution where two atomic blocks, b_1 and b_2, are cyclically dependent

To define this measure, we need an auxiliary definition: We say that atomic blocks b and b' are *cyclically dependent* in AADG G over trace τ, denoted by $b \overset{G}{\rightleftharpoons} b'$, if there are two memory locations l and l', and four transitions $(t_1, b) \prec (t_2, b) \prec (t_3, b') \prec (t_4, b')$ (where $\prec$ is the order of appearance of transitions in τ), such that

- t_1 and t_4 are the first transitions in b and b', respectively, to access l;
- t_2 and t_3 are the first transitions in b and b', respectively, to access l'; and
- $(t_1, b) \leftarrow (t_4, b')$ and $(t_2, b) \leftarrow (t_3, b') \in G.E$.

At runtime, this translates into a cyclic conflict if the executions of b and b' are interleaved (*i.e.* t_1 and t_3 are first executed, which forces cyclic dependence between b and b').

Based on the above definition, we quantify cyclic dependence as the following normalized measure:

$$\mathsf{cdep}\ G = 2 \cdot \left| \{ b \overset{G}{\rightleftharpoons} b' \} \right| / (\mathsf{nblks}\ G \cdot (\mathsf{nblks}\ G - 1)),$$

where $\mathsf{nblks}\ G$ is the number of distinct atomic blocks in G.

The algorithm in Figure 5 outputs a pair of real numbers in the range $[0, 1]$, which denote estimates of contention and cyclic dependencies. Together, these estimates provide a characterization of the available parallelism in trace τ, which we refer to as the *parallelism profile* of τ.

4. ADAPTATION VIA OFFLINE LEARNING

Given the mechanisms of Section 3 to estimate parallelism for an execution trace, we now describe an offline learning system that synthesizes—based on a finite number of "representative" traces for an application—a specialized adaptation strategy for that application. The learning process is independent of the parallelization system.

4.1 Learning per-input parallelism

A high-level view of the input-to-parallelism learning algorithm is given in Figure 7. This algorithm accepts as input a collection of sequential execution traces ($\tau^1 \ldots \tau^m$), along with their respective input workloads ($i^1 \ldots i^m$). TIGHTFIT derives a parallelism profile $\hat{p}^j$ from τ^j using quantitative analysis (by applying the o^τ algorithm explained in Section 3 to τ^j), and records the relationship between the features of input i^j and $\hat{p}^j$. (Recall that $\hat{p}^j$ estimates the available parallelism in τ^j.)

The connection between inputs and their respective parallelism profile is lifted—via regression analysis—into a predictor, o^f, from input features to an estimated parallelism profile, as shown in Figure 7: The free variables are the feature values, and the dependent variables are the estimates of contention and cyclic dependencies. By default, TIGHTFIT applies linear regression analysis, though the choice of regression model is parametric and configurable (*e.g.* cubic

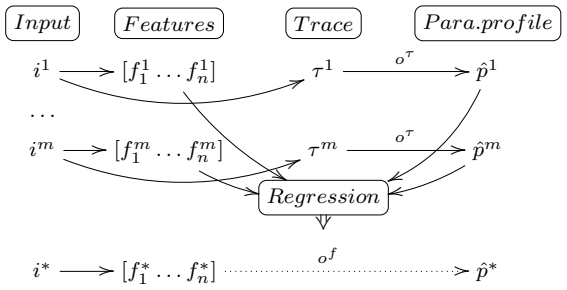

Figure 7: Abstract view of the offline input-to-parallelism learning alg.

or quartic regression). To make the mapping from inputs to parallelism profile amenable to learning, inputs are modeled according to the user-provided features.

Regression analysis makes statistical assumptions over its inputs. Specifically, the training set's size should be proportional to the number of independent variables, and the inputs should be sampled independently at random. To meet these requirements, TIGHTFIT provides a learning harness to the user. The user is asked to specify an admissible range of values for each application parameter. The harness then picks parameter values at random to craft training inputs.

By default, TIGHTFIT samples $\max\{150, 50m\}$ inputs, where m is the number of features. This is a conservative approximation of several popular guidelines, including $N \geq 104 + m$, $N \geq 40m$ and $N \geq 50 + 8m$ [9]. The user can set other bounds if desirable. Our experiments confirm these bounds as effective (*cf.* Section 6).

4.2 Threshold values for mode transitions

A remaining task after learning the predictor, o^f, is to correlate between parallelism estimates and modes of given adaptive parallelization systems. Building this additional bridge yields a direct relationship between input features and parallelization modes, which concludes the offline adaptation process with an adaptation strategy that chooses between system behaviors according to input features. We assume that the runtime adaptation system has several distinct modes, totally ordered according to parallelism level. Thus, correlating between parallelism profiles and parallelization modes is essentially the problem of setting thresholds for transitioning between modes.

TIGHTFIT computes the thresholds for every parallelization system, irrespectively of the particular application at hand. Similarly to other adaptive systems [38], this is achieved via a synthetic benchmark suite that enables parametric control over the amount of parallelism in a run. The benchmarks manipulate a shared `ConcurrentMap` object using a random client loop, where each iteration is an atomic task. The proportion of read/write operations, range of keys and number of operations are all parametric.

Within this setting, TIGHTFIT induces (a large number of) different parallelism profiles, checking for each which mode works best (*i.e.* results in shortest execution time). This yields empirical cutoff values for transitioning between modes as a function of the parallelism profile (TIGHTFIT's contention and cyclicity estimates).

Offline Learning of Adaptation Strategy

Inputs:
  $\{\tau_i\}_{i=1}^m$: execution traces with atomicity annotations
  features: feature extraction function

Parameters:
  π: adaptive para. system with modes $m_1 \prec \dots \prec m_k$
  mode: mapping from para. estimate $((D, C))$ to mode (m_i)
  regress: regression algorithm

OFFLINEADAPT: [returns actor function: $\{\overline{f} \mapsto \{m_i\}_{i=1}^k\}$]
  let $\pi = \emptyset$ [mapping from feature val.s to para. profiles]
  for each trace τ_i:
    let $\overline{f}_i$ = features σ_0 [σ_0 is the initial state in τ_i]
    let (D_i, C_i) = OFFLINEESTIMATE(τ_i)
    insert $\overline{f}_i \mapsto (D_i, C_i)$ into π
  let $o^f = \{\overline{f} \mapsto (\widetilde{D}, \widetilde{C})\}$ = regress π [avail. para. predictor]
  return $\{\overline{f} \mapsto (\text{mode} \cdot o^f)\ \overline{f}\}$ [actor: feature val.s to mode]

Figure 8: Offline alg. for synthesizing the "Actor" module for an adaptation scheme

4.3 Putting it all together

The complete learning system is shown in pseudocode in Figure 8, where mode is the thresholding algorithm. Note that in Figure 8, the offline learning algorithm is parameterized by the (system-specific) mapping from parallelism profiles to modes of the parallelization system π. The OFFLINEESTIMATE algorithm developed in Section 3 is a subroutine of OFFLINEADAPT. The output is the "Actor" module for parallelization system π, formed as the composition of mode and the parallelism predictor o^f.

This enables the runtime system outlined in Figure 1. The "Sensor" module realizes the reading of feature values off the application's state (initial state in general, and intermediate states to account for phase transitions), and the "Actor" module issues (if needed) a mode selection request based on the feature values sampled by the "Sensor": First, the "Actor" applies o^f to obtain a parallelism profile, thus estimating the available parallelism, and then it invokes the mode function to obtain the recommended mode.

5. ADAPTIVE RUNTIME SYSTEMS

Research on data-dependent parallelism has resulted in a wide range of adaptive concurrency control mechanisms. We have realized two such mechanisms to evaluate the efficacy of TIGHTFIT, which we describe in this section. In both cases, the initial adaptation decision is made at the beginning of a run. The user can configure TIGHTFIT to perform additional adaptation steps during the run to account for phase transitions. The user then also configures the frequency of adaptation decisions. A value n means that once every n transaction starts an adaptation decision will be taken.

5.1 Switching between STM protocols

Different STM protocols have different strengths and weaknesses. Some protocols work better under high contention, whereas others are specialized for high-parallelism workloads [24]. This leads to an adaptation method, whereby the runtime system switches between protocols according to the (estimated) available parallelism. We have designed such a system with three underlying protocols:

- The retry protocol (RETRY) applies eager specula-

tion [30], aborting a transaction immediately as it attempts conflicting access to a memory location "owned" by another live transaction (*i.e.* at least one of the transactions needs write access to the location). This works well for low-contention profiles.

- Dependence-aware transactional memory (DATM-FG) [25, 26] is effective for high-contention profiles with scarce cyclic dependencies between transactions. DATM-FG lets a transaction depend on another transaction "owning" a needed memory location, instead of aborting and retrying, effectively stalling the dependent transaction as long as no cyclic dependencies arise. This reduces concurrency, as well as retries, which is desirable for high-contention profiles.
- Finally, for high-contention profiles with a high proportion of cyclic dependencies, a coarse-grained variant of DATM-FG, dubbed DATM-CG, is reserved. This variant effectively simulates a global lock by viewing all memory locations as one and the same, which obviates the threat of rollbacks due to cyclic dependencies.

The resulting "Actor" module has the following form:

$$\mathsf{Actor}(\overline{f}) = \begin{cases} \textsc{Retry} & \text{if } \widetilde{con}(\overline{f}) \leq t_{con} \\ \text{DATM-FG} & \text{if } \widetilde{con}(\overline{f}) > t_{con}, \widetilde{cyc}(\overline{f}) \leq t_{cyc} \\ \text{DATM-CG} & \text{otherwise} \end{cases}$$

Because Retry and DATM-FG are friendly protocols [25], switching between these protocols is permitted even when there are live transactions synchronizing according to the old protocol. To switch to/from DATM-CG, however, a barrier is required. Otherwise serializability is not guaranteed. Our implementation instead makes an opportunistic attempt to enforce the selection, but gives up if the attempt fails due to live transactions. A backoff mechanism improves the probability that the next attempt will succeed.

5.2 Active threads

Another adaptation strategy is to set concurrency level—as dictated by the number of active threads—acrroding to the available parallelism [23, 15]. This is facilitated by parallelization systems with a configurable thread pool, whose size can be adjusted during the run, such as clients of the Java `ThreadPoolExecutor` library. The number of active threads can be adapted at any point in the run, without any constraints due to live transactions, and without affecting the correctness of the run.

In our implementation, the "Actor" follows the template

$$\mathsf{Actor}(\overline{f}) = \mathsf{ceil}(ncores \cdot \frac{\alpha \cdot \widetilde{con}(\overline{f}) + \beta \cdot \widetilde{cyc}(\overline{f})}{2}),$$

which yields an integral value in the range $[1, ncores]$ for real coefficients $\alpha, \beta \in [0, 1]$. That is, contention and cyclicity are given distinct weights in deciding the concurrency level, up to the number of available cores.

6. IMPLEMENTATION & EVALUATION

In this section, we describe our protoype implementation of the Tightfit system, and report on experiments measuring the effect of (offline) adaptation according to our approach in comparison with several other alternatives.

6.1 Prototype implementation

Tightfit is implemented as a Java library. Our prototype loads user specifications at runtime from a designated location. It additionally inserts instrumentation hooks at the beginning of code blocks designated for atomic execution (as specified by an annotation) to perform feature sampling and enforce adaptation requests. For instrumentation, we use the Javassist bytecode instrumentation library [6].

For offline analysis, our prototype exposes a configurable policy prescribing which regression algorithm to apply. We use algorithms from the Weka library [4] for regression analysis, the default being the `LinearRegression` class. The user can also control sampling parameters. For sampling, the default interval is 50 atomic blocks. We used these default settings in our experiments.

6.2 Experiments

Benchmarks. Our experimental suite included eight benchmarks. These are described in Table 1. All benchmarks, excluding Boruvka and Elevator, were taken from JSTAMP—the Java port of the STAMP benchmark suite [18]—which is distributed as part of the Deuce STM implementation [1]. The Boruvka benchmark is based on a sequential implementation of Boruvka's algorithm that is available as part of the Java version of the LoneStar suite [13]. Elevator is taken from the pjbench suite of parallel Java benchmarks [2]. Elevator and Bank are lock-based parallel applications. MatrixMultiply is embarrassingly parallel, admitting zero conflicts on any input matrix, but is included in the evaluation for sanity checking and overhead assessment. The remaining five benchmarks are irregular, and thus use STM.

Our specification of workload features over these benchmarks was straightforward: For all benchmarks except Boruvka, we set the command-line arguments as the candidate features. This is because the benchmarks are all parametric per command-line values. For Boruvka, we specified the three features in Figure 4; namely: number of nodes, density and average node degree.

Experimental setup. We designed two experiments, corresponding to the adaptive systems described in Section 5. In the first experiment, we implemented an adaptive system that can switch between STM protocols (*cf.* Section 5.1), and used the Boruvka algorithm as well as the five JSTAMP STM applications as benchmarks. In the second experiment, we investigated adaptation by means of changing concurrency levels (*cf.* Section 5.2). This mechanism works uniformly both for STM and for lock-based synchronization, and thus we considered all eight benchmarks. (The STM benchmarks used the "DATM-FG" protocol.)

In the first experiment, we compared Tightfit with three non-adaptive STMs, each using one of the protocols underlying the adaptive system ("Retry", "DATM-FG" and "DATM-CG"). In the second experiment, we compared Tightfit against three fixed concurrency levels ("2 thr.s", "4 thr.s" and "8 thr.s", where "1 thr" serves as a reference), as well as an online adaptation algorithm ("Online") that switches between the protocols by tracking per-thread abort/commit ratio [5, 40].

In both experiments, we also compared the efficacy of the two-step offline learning technique featured in Tightfit (where the application-specific predictor is computed separately from the system-specific thresholding function) with two offline adaptation algorithms ("Off-4" and "Off-8") that learn a predictor from input features to system modes directly. This is achieved by correlating between input features and the mode yielding the shortest execution time for

Benchmark	Domain/Description	Inputs
Boruvka	Scientific: Computes MST	5000–10000 nodes, avg. degree between 1–5
Genome	Bioinformatics: Performs gene sequencing	$256 \leq -\texttt{g} \leq 1024$, $8 \leq -\texttt{s} \leq 64$, $32768 \leq -\texttt{n} \leq 131072$
Intruder	Security: Detects network intrusions	$1 \leq -\texttt{a} \leq 25$, $2 \leq -\texttt{l} \leq 32$, $1024 \leq -\texttt{n} \leq 65536$, $1 \leq -\texttt{s} \leq 256$
KMeans	Data mining: Implements K-means clustring	$5 \leq -\texttt{m}, -\texttt{n} \leq 320$, $100^{-1} \leq -\texttt{s} \leq 4^{-1}$
MatrixMultiply	Scientific: Performs matrix multiplication	$N \times N$ matrices, where $100 \leq N \leq 1000$
Vacation	Online tx. processing: Emulates a travel reservation sys.	$1 \leq -\texttt{n} \leq 10$, $1 \leq -\texttt{q}$, $-\texttt{u} \leq 100$,$2^{13} \leq -\texttt{r} \leq 2^{15}$, $2^{10} \leq -\texttt{t} \leq 2^{13}$
Bank	Online tx. processing: Emulates a banking sys.	$1000 \leq -\texttt{n} \leq 10000$, $0 \leq -\texttt{i} \leq 50000$,$1000 \leq -\texttt{m} \leq 100000$
Elevator	Discrete event simulator: Simulates a sys. of elevators	10–10000 floors, random bias over from/to floor numbers

Table 1: Benchmarks, including parameter ranges for random workload generation

that input, where "Off-4" utilizes four threads for this task and "Off-8" trains with eight threads.

We conducted the experiments on a 64-bit Linux machine with an Intel Core i7-870 processor combining four dual cores (at 2.93GHz), each multiplexing 2 hardware threads, for a total of 8 threads. The host VM was Java SE Development Kit 6 Update 25 (JDK6u25). The input ranges we used, both for offline analysis and for the deployment runs, are listed in column three of Table 1. The number of traces we considered for training is discussed in Section 4.1. For the production runs, we selected at random 20 inputs per benchmark. For each input and concurrency level, ranging from 1 to 8 threads, we ran the benchmark 4 times and recorded the average across the last 3 runs, excluding the first (cold) run. The measurements reported in Section 6.3 represent the average across all 20 of the selected inputs.

6.3 Performance results

Speedup and retry trends for the first experiment (the system of Section 5.1), going from one to eight threads, are shown in Figures 9–14 and Figures 16–20, respectively. We omit retry statistics for MatrixMultiply, which does not admit any conflicts (*cf.* Section 6.2). We also omit retry statistics for the "DATM-CG" protocol, since this protocol simulates a global lock, and thus prevents retries completely.

Average speedup and retries for eight threads are summarized below:

	Speedup		Retries	
	all	wo. MMul	all	wo. MMul
Retry	3.75	3.04	1.53	1.84
DATM-FG	4.38	3.77	0.32	0.38
DATM-CG	3.96	3.28	—	—
TIGHTFIT	4.91	4.43	0.21	0.25
Online	4.18	3.54	0.52	0.62
Off-4	4.92	4.44	0.22	0.26
Off-8	5.27	4.83	0.19	0.22

TIGHTFIT achieves better speedup, and less retries, than the fixed alternatives (*i.e.* "Retry", "DATM-FG" and "DATM-CG") as well as online adaptation ("Online"). As for direct offline learning ("Off-4" and "Off-8"), at first glance the results seem to suggest that these are comparable if not superior to TIGHTFIT: "Off-4" has similar speedup and retry statistics to TIGHTFIT over eight threads, and "Off-8" is approximately 8% faster (more accurately: 7% with MatrixMultiply and 9% without MatrixMultiply). However, more careful analysis of the results—and in particular, the speedup and retry trends visualized in Figures 9–14 and Figures 16–20, respectively—reveals that on average, TIGHTFIT is as effective as "Off-4" and "Off-8" on benchmarks with high variance in parallelism, such as Genome and KMeans, *across the entire range of concurrency levels*. We return to this point in Section 6.4.

The speedup results we obtained in the second experiment (the system of Section 5.2) are listed in Figure 22. For some of the benchmarks (including Elevator, KMeans and Genome), TIGHTFIT is superior to alternatives fixing the number of active threads throughout all inputs and the entire duration of the run. Retries and memory usage statistics for the lock-based benchmarks as well as nonzero-retry STM benchmarks under "DATM-FG" are reported below:

Mode	Retries			Memory	
	Genome	*Boruvka*	*Vacation*	*Bank*	*Elevator*
1 thread	0	0	0	1	1
2 threads	0.18	0.07	0.19	0.98	0.99
4 threads	0.22	0.2	0.48	0.95	0.96
8 threads	0.56	0.46	0.99	0.92	0.94
TIGHTFIT	0.47	0.31	0.76	0.93	0.94
Off-4	0.53	0.36	0.70	0.94	0.95
Off-8	0.51	0.33	0.72	0.96	0.96

Again here, "Off-4" and "Off-8" appear slightly better than TIGHTFIT on some benchmarks and concurrency levels, though consistently with the results of the first experiment, TIGHTFIT achieves comparable speedups across all concurrency levels on benchmarks with highly dynamic parallelism (namely, Boruvka, Genome and KMeans).

6.4 Discussion

Overall, the experimental results provide support for the main thesis of this paper, which puts forward the direct connection between input features and available parallelism as the basis for adaptation. For the adaptive STM system (Section 5.1), TIGHTFIT's offline adaptation algorithm is measurably better than its online alternative, also yielding better results than the three baseline STM algorithms. A similar trend is seen with the system of Section 5.2, where adaptation is achieved by controlling concurrency level.

Moreover, our analysis of the offline artifacts confirms that the learning algorithm is able to converge on the features that effectively control parallelism. As an example, TIGHTFIT was able to converge on `-q` and `-n` as the significant features in Vacation. These two parameters control transaction duration and query range, respectively, and are indeed the determining factors of available parallelism in Vacation (where `-r` and `-t` are global parameters affecting the overall duration of the run), as the documentation for this benchmark confirms.

The improvement thanks to TIGHTFIT is more noticeable on benchmarks whose available parallelism is highly dynamic, such as KMeans and Genome, and to a lesser extent also Boruvka. Benchmarks with less variance in parallelism profiles, such as Vacation, provide less room for optimization

thanks to (offline) adaptation, leaving TIGHTFIT similar to the "Online" adaptation algorithm in performance.

This observation also connects to the comparison between TIGHTFIT and the two other offline adaptation techniques: "Off-4" and "Off-8". While benchmarks whose available parallelism is more stable across different workloads seem to favor "Off-4" and "Off-8", TIGHTFIT is competitive with both of these alternatives on the benchmarks whose parallelism changes significantly across inputs throughout the entire spectrum of concurrency levels. In particular, while "Off-4" is more effective in the neighborhood of four threads, and"Off-8" performs better when concurrency level is closer to eight threads, TIGHTFIT is more stable than both of these alternatives, and thus achieves comparable speedups on average across all concurrency levels.

The experiments indicate the superiority of the offline adaptation technique suggested in this paper. However, it appears that if the deployment setup (*i.e.* concurrency level, cache sizes, processor architecture, etc) is known *a priori*, then direct learning (in the style of "Off-4" and "Off-8") is potentially preferable. If, on the other hand, the deployment setup is subject to variation (*e.g.* due to usage of several different hardware configurations), then we expect TIGHTFIT's learning algorithm to provide better results as it abstracts away all deployment details.

Another advantage of the offline learning algorithm of TIGHTFIT over direct learning is that it allows for separation of concerns: The application designer provides useful input features, and the designer of the adaptive system separately decides which execution mode best fits every parallelism profile. Recall that TIGHTFIT automatically learns the mapping from parallelism profiles to system modes. We believe that if this mapping could be specified by an expert, then TIGHTFIT would achieve even better results. (Asking for such a specification is reasonable, as it is done once per system.) We leave research on this topic for future work.

7. RELATED WORK

We discuss related research in three categories: online adaptation; offline parallelization optimizations, including user interaction; and tools for estimating parallelism. In the broader scope of profile-guided specialization, we refer the reader to [27] for adaptive garbage collection, [33] for profile-guided compile-time parallelization, and [19] for profile-based specialization of static heap abstractions.

Online adaptation.

There are various proposals how to detect and respond to changes in available parallelism online, during parallel execution. The *transactional concurrency tuning* system [5] uses a control-theoretic model to adapt the number of active threads to the available parallelism, where the percentage of committed transactions out of all executed transactions in a sample period provides a measure of available parallelism. A closely related approach, presented in [40], is to stall a thread if its abort/commit history indicates low parallelism. A similar heuristic is implemented in the Galois system [15]. The main distinction from our approach is that TIGHTFIT estimates available parallelism directly, based on input features, instead of tracking indirect monitors related to the execution system's behavior.

The Shrink system [8] utilizes the run prefix to predict memory accesses. The predicted transactional accesses are based on the access patterns of past transactions from the same thread. The scheduler then tries to prevent transactions whose predicted access sets are in overlap from running in parallel. Tracking memory acccesses is reminiscent of our analysis of data dependencies, though Shrink features online adaptation. It is not clear how to cast Shrink into our setting of offline learning.

Another form of adaptation, proposed in [36], is adaptive locking: The parallelization algorithm monitors the execution, and—based on the collected statistics—decides whether to execute a critical section (CS) speculatively or using mutual exclusion. In [28], "hot" variables, which cause large numbers of transactions to abort, are identified at runtime; for these variables, the transactional manager selectively switches to pessimistic concurrency control, where reader/writer locks mediate access to locations.

adaptSTM [24, 20] utilizes runtime parameters like the number of unique read and write locations, the number of hashtable collisions and the ratio of aborts to commits to tune STM internal data structures, such as the size of the hashtable mapping memory to locks and the choice of hash function. The system of [29] goes beyond such fine-grained adaptivity (that optimizes a specific STM implementation) to also support coarse-grained adaptivity, where a system-wide policy specifies when to switch between STM implementations.

The Sambamba system [31]—designed to automatically optimize legacy C-like programs—monitors the subject application's behavior and specializes functions on-the-fly for actual usage profiles. One of the supported optimizations is automatic parallelization using speculation, where the decision whether to apply parallelization is dependent on the available compute resources.

All of these systems currently adapt their behavior online, and could benefit from integration with TIGHTFIT, so that the choice of parallelization mode (*e.g.* configuration of internal data structures in adapSTM or CS execution mode in adaptive locking) is made offline based on input characteristics.

Hybrid techniques.

A mechanism for dynamic profiling of a running transactional program is presented in [38]. The obtained profile is then used, in conjunction with a machine-learning (ML) algorithm, to select an "optimal" STM implementation at runtime. The ML algorithm is trained offline by measuring the running time of parameterized microbenchmarks for all available STM algorithms at multiple thread levels. Then, during program execution, a fixed number of transactions is profiled to guide the ML algorithm's choice of STM algorithm. The offline learning phase records certain code-level characteristics of the microbenchmarks, such as whether transactions contain loops. The online prediction rule is parameterized by these static distinctions. Recall that TIGHTFIT estimates the available parallelism as a function of user-specified properties of the data. In contrast, the predictor in [38] relies on predefined syntactic features.

In [32], the authors utilize offline analysis to discover *seminal behaviors* These are behaviors that typically manifest at an early stage of execution (*e.g.* the values of certain program variables or the correlation between the number of iterations of different loops), and are correlated with many other behaviors of the program, which enables effective pre-

diction of the program's behavior. Seminal behaviors are extracted automatically, as an approximation of input characteristics to enable proactive optimizations in the Jikes VM. A similar approach, developed in [11], trades training for incremental adaptation across production runs. The authors of [11] apply this idea to predict the likelihood of successful speculation, where predictions account for input properties indirectly, through the automatic learning technique of classification trees. TIGHTFIT shares the motivation of these works, but supports direct learning of the relationship between input features and parallelism, rather than passing through seminal behaviors, thanks to offline learning.

The Janus algorithm [34] uses offline training to improve the precision of conflict detection during speculative parallelization. The key idea in Janus is to enforce sequence-based detection, where sequences of operations—rather than individual operations—are tested for commutativity. The training phase is used to observe which sequences occur in the client application and check offline whether they commute, so that the runtime overhead of sequence-based detection becomes negligible. TIGHTFIT is similar to Janus in applying offline learning, but the learning process—including its scope, flow and techniques—is quite different.

Offline and interactive prediction.

The analysis algorithm in [37] computes *dependence densities* for pairs of tasks that are intended for parallel execution, where dependence density is a measure of the probability that two randomly chosen tasks from the same computation phase are data dependent. The analysis is based on deterministic single-threaded program runs. The tool provides program-wide (rather than input-sensitive) insight into the potential for optimistic parallelization, opportunities for algorithmic scheduling, and performance defects due to synchronization bottlenecks. Dependence densities share some similarity with TIGHTFIT's offline analysis, though we compute different quantitative measures, as well as structural properties of dependencies, such as cyclic dependencies.

An interactive tool for analyzing the behavior of a transactional program, and in particular, the causes and magnitude of wasted work caused by aborting transactions, is presented in [41]. The tool learns about potential conflicts from dynamic runs. It identifies the data structures involved in a conflict using a symbolic path through the heap, and provides visualization techniques to summarize how threads spend their time and which transactions conflict most frequently. The tool also makes recommendations how to optimize the parallel program using known techniques, such as nested transactions, checkpoints and early release.

The Hawkeye tool [35] assists in interactive parallelization by analyzing traces from sequential runs of the subject application and identifying *semantic* conflicts between computations that the user wishes to run concurrently. This is accomplished by augmenting the standard approach of dependence analysis with abstraction specifications for shared objects (formulated as representation functions).

The framework of [7] leverages programmer knowledge about data-structure semantics to decide whether to throttle concurrency when executing a program using STM, where the developer is asked to provide predicates describing the semantic footprint of an operation (*e.g.* by asserting that two arguments are not aliased). The STM then throttles the starts of transactions that have a high probability of conflict with active transactions. The SAW system [39] uses aspects to decouple the synchronization mechanism from the application's logic. The developer is responsible to select a suitable synchronization mechanism for a critical section: either locking or STM. The user specifies critical sections and shared objects using SAW annotations.

A scheme based on compiler analysis for identifying static atomic sections whose instances, when executed concurrently by more than one thread, necessarily conflict is described in [16]. To reduce the overhead due to *always conflicting atomic sections* (ACASs), the authors propose two techniques: *selective pessimistic concurrency control*, where the level of concurrency is reduced when executing ACASs, and *early conflict checks*, which enforces more eager conflict checking to minimize the amount of wasted work in the event of a conflict. Another static analysis is [22], which utilizes shape analysis to optimize optimistically parallel graph programs. The analysis identifies fail-safe points in the program, beyond which execution is guaranteed not to abort and backup copies of modified data are not needed. The analysis further eliminates redundant conflict checking.

The ParaMeter [14] algorithm produces *parallelism profiles* for irregular programs, where a parallelism profile is a measure of the amount of amorphous data parallelism at different points in the execution of the parallel algorithm. This measure, similarly to our analysis of data dependencies, abstracts away implementation-specific details, such as the number of cores, cache sizes and load balancing. ParaMeter can also generate constrained parallelism profiles for a fixed number of cores.

The main distinction between the above works and our approach is that these analyses aim for general characterization of the application's available parallelism, where TIGHTFIT specializes in building a mapping function from input features to available parallelism. Moreover, TIGHTFIT utilizes profiling information for actual parallelization.

8. CONCLUSION AND FUTURE WORK

We have presented a novel approach for foresight-guided adaptation, which permits low-overhead, input-centric runtime adaptation by shifting most of the cost of predicting per-input parallelism to an expensive offline analysis. Two aspects of our approach are of particular interest: (i) specification of workloads in terms of useful features, which permits direct learning of the connection between input characteristics and available parallelism, and (ii) quantitative and structural analysis of data dependencies as a means of estimating available parallelism while abstracting away deployment-specific details. Our approach is implemented in the TIGHTFIT system, which is publicly available [3], and shows promising results in our experiments.

In the future, we intend to make TIGHTFIT more automatic by introducing auto-tuning capabilities, similarly to [10] (*e.g.* to decide on effective threshold values for mode transitions). We also plan to develop inference capabilities to automatically converge on useful workload features.

9. REFERENCES

[1] Deuce stm. http://www.deucestm.org.

[2] The pjbench suite. http://code.google.com/p/pjbench/.

[3] Tightfit. http://www.cs.tau.ac.il/~omertrip/software/tightfit/tightfit.html.

[4] The weka library. http://sourceforge.net/projects/weka.

[5] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris C. Kirkham, and Ian Watson. Robust adaptation to available parallelism in transactional memory applications. *T. HiPEAC*, 3, 2011.

[6] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *GPCE*, 2003.

[7] R. Cledat, K. Ravichandran, and S. Pande. Leveraging data-structure semantics for efficient algorithmic parallelism. In *ICCF*, 2011.

[8] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *PODC*, 2009.

[9] P. I. Good and J. W. Hardin. Common errors in statistics (and how to avoid them). *The American Statistician*, 58, 2004.

[10] P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv. Concurrent data representation synthesis. In *PLDI*, 2012.

[11] Y. Jiang, F. Mao, and X. Shen. Speculation with little wasting: Saving cost in software speculation through transparent learning. In *ICPDS*, 2009.

[12] Y. Jiang, E. Z. Zhang, K. Tian, F. Mao, M. Gethers, X. Shen, and Y. Gao. Exploiting statistical correlations for proactive prediction of program behaviors. In *CGO*, 2010.

[13] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS*, 2009.

[14] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? In *PPOPP*, 2009.

[15] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[16] S. Mannarswamy and R. Govindarajan. Handling conflicts with compiler's help in software transactional memory systems. In *ICPP*, 2010.

[17] F. Mao and X. Shen. Cross-input learning and discriminative prediction in evolvable virtual machines. In *CGO*, 2009.

[18] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

[19] M. Naik, H. Yang, G. Castelnuovo, and M. Sagiv. Abstractions from tests. In *POPL*, 2012.

[20] M. Payer and T. Gross. adaptSTM - an online fine-grained adaptive stm system. Technical report, ETH Zurich, 2010.

[21] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *PLDI*, 2011.

[22] D. Prountzos, R. Manevich, K. Pingali, and K. S. McKinley. A shape analysis for optimizing parallel graph programs. In *POPL*, 2011.

[23] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In *IISWC*, 2011.

[24] M. Payer T. R. and Gross. Performance evaluation of adaptivity in software transactional memory. In *ISPASS*, 2011.

[25] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *proc.s of the 41st annual IEEE/ACM intl. Symp. on Microarchitecture*, 2008.

[26] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an stm. In *PPOPP*, 2009.

[27] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *ISMM*, 2004.

[28] N. Sonmez, T. Harris, A. Cristal, O. S. Unsal, and M. Valero. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. In *IPDPS*, 2009.

[29] M. F. Spear. Lightweight, robust adaptivity for software transactional memory. In *SPAA*, 2010.

[30] M. F. Spear, V. J. Marathe, W. N. Scherer, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *ICDC*, 2006.

[31] K. Streit, C. Hammacher, A. Zeller, and S. Hack. Sambamba: A runtime system for online adaptive parallelization. In *Compiler Construction*, 2012.

[32] K. Tian, Y. Jiang, E. Z. Zhang, and X. Shen. An input-centric paradigm for program dynamic optimizations. In *OOPSLA*, 2010.

[33] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI*, 2009.

[34] O. Tripp, M. Manevich, J. Field, and M. Sagiv. Janus: Exploiting parallelism via hindsight. In *PLDI*, 2012.

[35] O. Tripp, G. Yorsh, J. Field, and M. Sagiv. Hawkeye: effective discovery of dataflow impediments to parallelization. In *OOPSLA*, 2011.

[36] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis. Adaptive locks: Combining transactions and locks for efficient concurrency. In *PACT*, 2009.

[37] C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *PPOPP*, 2008.

[38] Q. Wang, S. Kulkarni, J. Cavazos, and M. Spear. A transactional memory with automatic performance tuning. *ACM Transactions on Architecutre and Code Optimization*, 8, 2012.

[39] Y. Yamada, H. Iwasaki, and T. Ugawa. Saw: Java synchronization selection from lock or software transactional memory. *ICPDS*, 0, 2011.

[40] R. M. Yoo and H. H. Lee. Adaptive transaction scheduling for transactional memory systems. In *proc.s of the twentieth annual Symp. on Parallelism in algorithms and architectures*, 2008.

[41] F. Zyulkyarov, S. Stipic, O. S. Unsal, A. Cristal, T. Harris, I. Hur, and M. Valero. Profiling and optimizing transactional memory applications. *Intl. Journal of Parallel Programming*, 40, 2012.

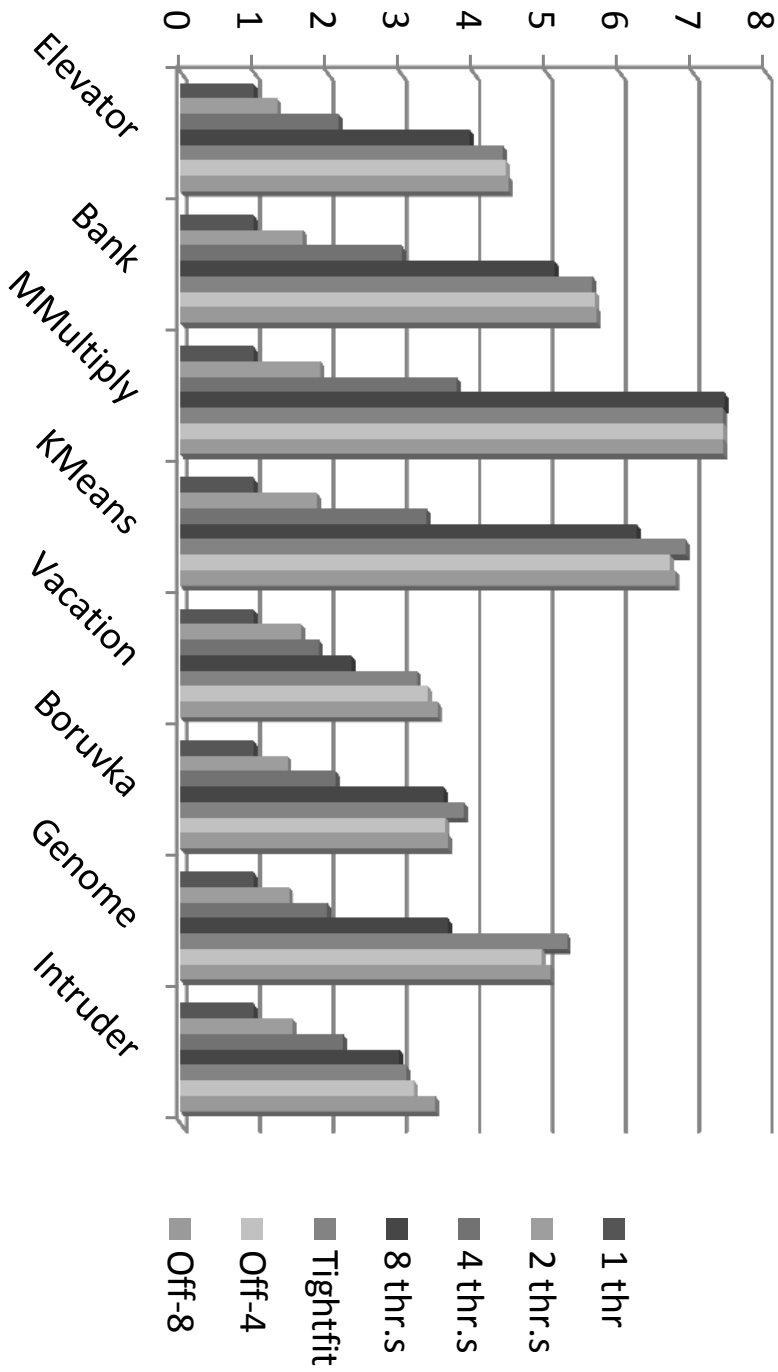

Figure 22: Speedup as function of active threads

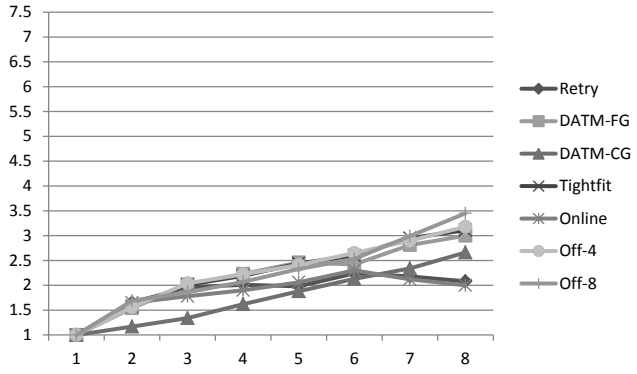

Figure 9: Intruder

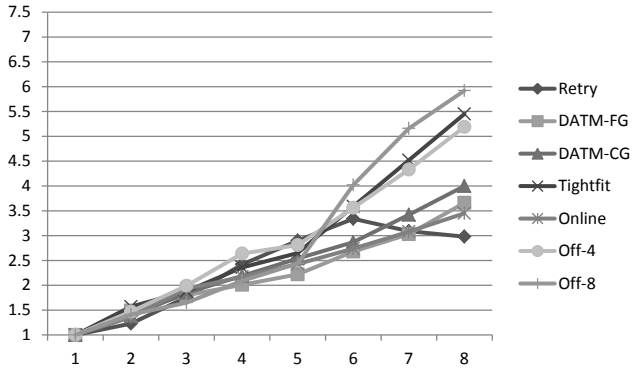

Figure 10: Genome

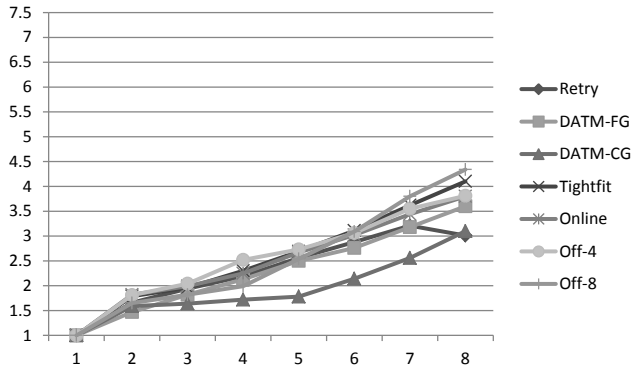

Figure 11: Boruvka

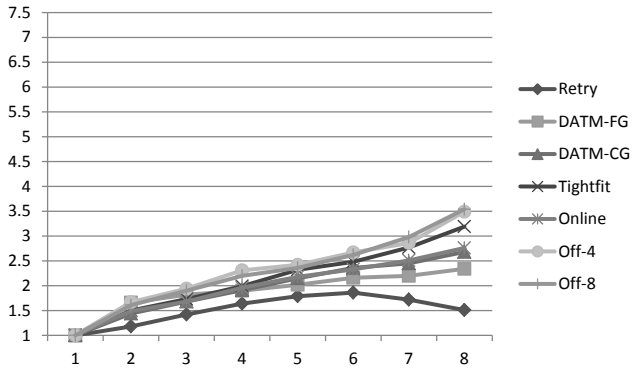

Figure 12: Vacation

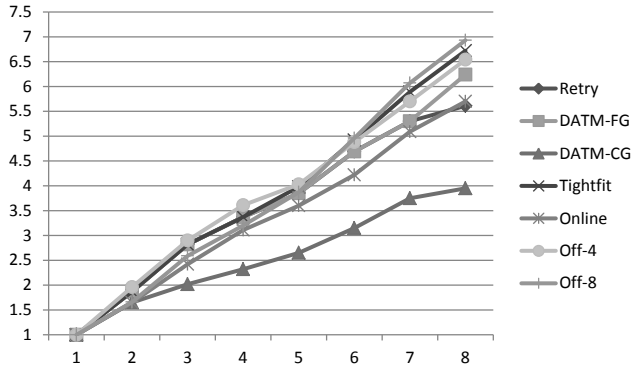

Figure 13: KMeans

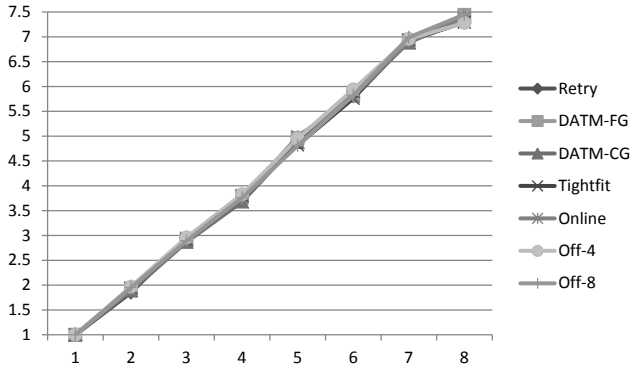

Figure 14: MatrixMultiply

Figure 15: Speedup trends moving from 1 to 8 threads

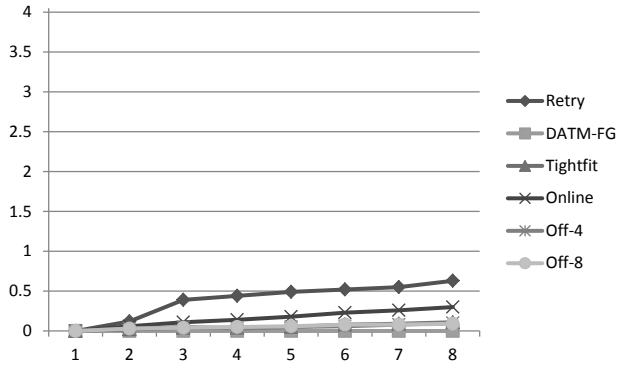

Figure 16: Intruder

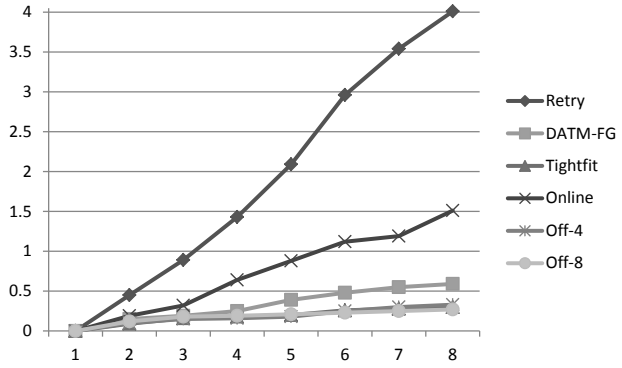

Figure 17: Genome

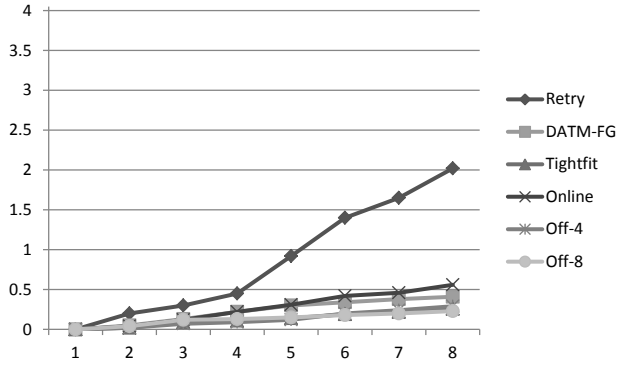

Figure 18: Boruvka

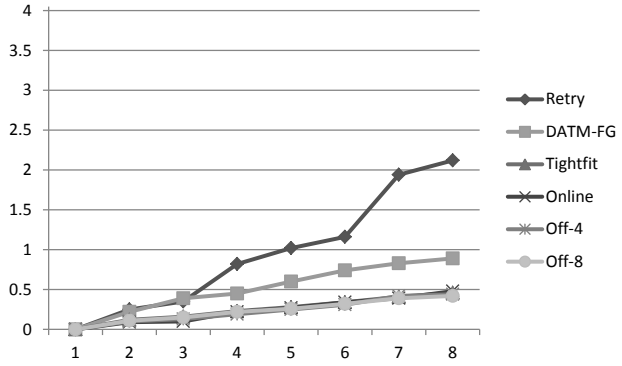

Figure 19: Vacation

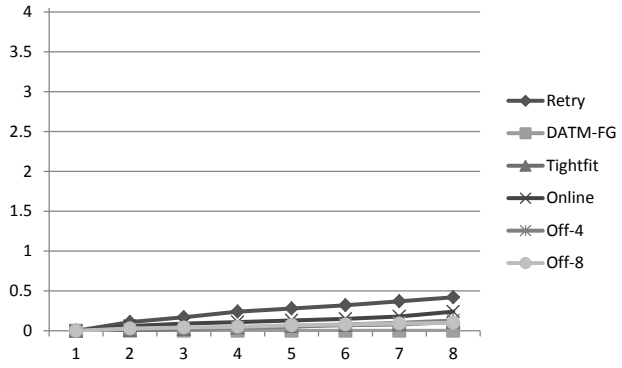

Figure 20: KMeans

Figure 21: Average number of retries per transaction moving from 1 to 8 threads