AutoSynch: An Automatic-Signal Monitor Based on Predicate Tagging

Wei-Lun Hung Vijay K. Garg

Department of Electrical and Computer Engineering
The University of Texas at Austin
wlhung@utexas.edu, garg@ece.utexas.edu

Abstract

Most programming languages use monitors with *explicit signals* for synchronization in shared-memory programs. Requiring programmers to signal threads explicitly results in many concurrency bugs due to missed notifications, or notifications on wrong condition variables. In this paper, we describe an implementation of an automatic signaling monitor in Java called *AutoSynch* that eliminates such concurrency bugs by removing the burden of signaling from the programmer. We show that the belief that automatic signaling monitors are prohibitively expensive is wrong. For most problems, programs based on *AutoSynch* are almost as fast as those based on explicit signaling. For some, *AutoSynch* is even faster than explicit signaling because it never uses *signalAll*, whereas the programmers end up using *signalAll* with the explicit signal mechanism.

AutoSynch achieves efficiency in synchronization based on three novel ideas. We introduce an operation called *closure* that enables the predicate evaluation in every thread, thereby reducing context switches during the execution of the program. Secondly, *AutoSynch* avoids *signalAll* by using a property called *relay invariance* that guarantees that whenever possible there is always at least one thread whose condition is true which has been signaled. Finally, *AutoSynch* uses a technique called *predicate tagging* to efficiently determine a thread that should be signaled. To evaluate the efficiency of *AutoSynch*, we have implemented many different well-known synchronization problems such as the producers/consumers problem, the readers/writers problems, and the dining philosophers problem. The results show that *AutoSynch* is almost as efficient as the explicit-signal monitor and even more efficient for some cases.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; classes and objects; control structures

General Terms Algorithms, Languages, Performance

Keywords automatic signal, explicit signal, implicit signal, monitor, concurrency, parallel

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

1. Introduction

Multicore hardware is now ubiquitous. Programming these multicore processors is still a challenging task due to bugs resulting from concurrency and synchronization. Although there is widespread acknowledgement of difficulties in programming these systems, it is surprising that by and large the most prevalent methods of dealing with synchronization are based on ideas that were developed in early 70's [2, 9, 14]. For example, the most widely used threads package in C++ [19], pthreads [6], and the most widely used threads package in Java [11], java.util.concurrent [18], are based on the notion of monitors [2, 14](or semaphores [8, 9]). In this paper, we propose a new approach, *AutoSynch*, based on automatic signaling monitor that allows gains in productivity of the programmer as well as gain in performance of the system.

Both pthreads and Java require programmers to explicitly signal threads that may be waiting on certain condition. The programmer has to explicitly declare condition variables and then signal one or all of the threads when the associated condition becomes true. Using the wrong waiting notification (*signal* versus *signalAll* or notify versus notifyAll) is a frequent source of bugs in Java multithreaded programs. In our proposed approach, *AutoSynch*, there is no notion of explicit condition variables and it is the responsibility of the system to signal appropriate threads. This feature significantly reduces the program size and complexity. In addition, it allows us to completely eliminate signaling more than one thread resulting in reduced context switches and better performance. The idea of automatic signaling was initially explored by Hoare in [14], but rejected in favor of condition variables due to efficiency considerations. The belief that automatic signaling is extremely inefficient compared to explicit signaling is widely held since then and all the prevalent concurrent languages based on monitors use explicit signaling. For example, Buhr, Fortier, and Coffin claim that automatic monitors are 10 to 50 times slower than explicit signals [4]. We show in this paper that the widely held belief is wrong. The reason for this drastic slowdown in previous implementations of automatic monitor is that they evaluate all possible conditions on which threads are waiting whenever the monitor becomes available. With careful analysis of the conditions on which the threads are waiting and evaluating as few conditions as possible, automatic signaling can be as efficient as explicit signaling. In *AutoSynch*, the programmer simply specifies the predicate P on which the thread is waiting by using the construct *waituntil(P)* statement. When a thread executes the *waituntil* statement, it checks whether P is true. If it is true, the thread can continue; otherwise, the thread must wait for the system to signal it. The *AutoSynch* system has a condition manager that is responsible for determining which thread to signal by analyzing the predicates and the state of the shared object.

Fig. 1 shows the difference between the Java and the *AutoSynch* implementation for the parameterized bounded-buffer problem, a variant bounded-buffer problem (also known as the producer-consumer problem) [8, 10]. In this problem, producers put items into a shared buffer, while consumers take items out of the buffer. The *put* function has a parameter $items$; the *take* function has a parameter, *num*, indicating the number of items taken. There are two requirements for synchronization. First, every operation on a shared variable, such as *buff*, should be done under mutual exclusion. Second, we need *conditional synchronization*; a producer must wait when the buffer does not have sufficient space, and a consumer must wait when the buffer has no sufficient items. The explicit-signal bounded-buffer is written in Java. Programmers need to explicitly associate conditional predicates with condition variables and call *signal* (*signalAll*) or *await* statement manually. Note that, the *unlock* statement should be done in a *finally* block, *try* and *catch* blocks are also need for the *InterruptedException* that may be thrown by *await*. However, for simplicity, we avoid the exception handling in Fig. 1. The automatic-signal bounded-buffer is written using *AutoSynch* framework. We use *AutoSynch* modifier to indicate that the class is a monitor as in line 1. An *AutoSynch* class provides mutually exclusive access to its member functions. For conditional synchronization, we use *waituntil* as in line 9. There are no *signal* or *signalAll* calls in the *AutoSynch* program. Note that, the *waituntil* statement can take any boolean condition just like the *if* and *while* statements. Clearly, the automatic-signal monitor is much simpler than the explicit-signal monitor.

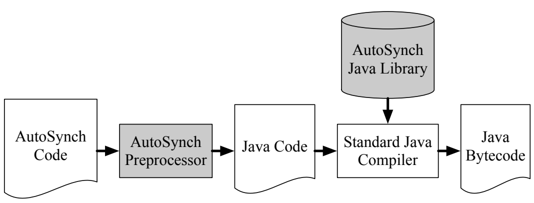

Figure 2: The framework of *AutoSynch*

The framework for the implementation is shown if Fig. 2. It is composed of a preprocessor and a Java condition manager library. The preprocessor translates *AutoSynch* code into traditional Java code. Our automatic-signal mechanism and developed techniques were implemented in the Java condition manager library, which is responsible for monitoring the state of the monitor object and signaling an appropriate thread.

In this paper, we argue that automatic signaling is generally as fast as explicit signaling (and even faster for some examples). The explicit signaling has to resort to *signalAll* in some examples; however, our automatic signaling never uses *signalAll*. Thus *AutoSynch* is considerably faster for synchronization problems that requires *signalAll*. The design principle underlying *AutoSynch* is to reduce the number of context switches and predicate evaluations.

Context switch: A context switch requires a certain amount of time to save and load registers and update various tables and lists. Reducing unnecessary context switches boosts the performance of the system. A *signalAll* call introduces unnecessary context switches; therefore, *signalAll* calls are never used in *AutoSynch*.

Predicate evaluation: In the automatic-signal mechanism, signaling a thread is the responsibility of the system. The number of predicate evaluations is crucial for efficiency in deciding which thread should be signaled. By analyzing the structure of the predicate, our system reduces the number of predicate evaluations.

There are three important novel concepts in *AutoSynch* that enable efficient automatic signaling — *closure of predicates, relay invariance*, and *predicate tagging*.

The technique of *closure* of a predicate P is used to reduce the number of context switches for its evaluation. In the current systems, only the thread that is waiting for the predicate P can evaluate it. When the thread is signaled, it wakes up, acquires the lock to the monitor and then evaluates the predicate P. If the predicate P is false, it goes back to wait resulting in an additional context switch. In *AutoSynch* system, the thread that is in the monitor evaluates the condition for the waiting thread and wakes it only if the condition is true. Since the predicate P may use variables local to the thread waiting on it, *AutoSynch* system derives a closure predicate P' of the predicate P, such that other threads can evaluate P'. The details of closure are in Section 3.1.

The idea of *relay invariance* is used to avoid *signalAll* calls in *AutoSynch*. The relay invariance ensures that if there is any thread whose waiting condition is true, then there exists at least one thread whose waiting condition is true and is signaled by the system. With this invariance, the *signalAll* call is unnecessary in our automatic-signal mechanism. This mechanism reduces the number of context switches by avoiding *signalAll* calls. The details of this approach are in Section 3.2.

The idea of *predicate tagging* is used to accelerate the process of deciding which thread to signal. All the waiting conditions are analyzed and tags are assigned to every predicate according to its semantics. To decide which thread should be signaled, we identify tags that are most likely to be true after examining the current state of the monitor. Then we only evaluate the predicates with those tags. The details of predicate tagging are in Section 3.3.

Our experimental results indicate that *AutoSynch* can significantly improve performance compared to other automatic-signal mechanisms [5]. In [4, 5] the automatic-signal mechanism is 10-50 times slower than the explicit-signal mechanism; however, *AutoSynch* is only 2.6 times slower than the explicit-signal mechanism even in the worst case of our experiment results. Furthermore, *AutoSynch* is 26.9 times faster than the explicit-signal mechanism in the parameterized bounded-buffer problem that relies on *signalAll* calls. Besides, the experimental results also show that *AutoSynch* is scalable; the performance of *AutoSynch* scales well even if the number of threads increases for many problems studied in the paper.

Although the experiment results show that *AutoSynch* is 2.6 times slower than the explicit-signal mechanism in the worst case, it is still desirable to have automatic signaling. First, automatic signaling simplifies the task of concurrent programming. In explicit-signal monitor, it is the responsibility of programmers to explicitly invoke a *signal* call on some condition variable for conditional synchronization. Using the wrong notification, and signaling a wrong condition variable are frequent sources of bugs. The idea is analogous to automatic garbage collection. Although garbage collection leads to decreased performance because of the overhead in deciding which memory to free, programmers avoid manual memory deallocation. As a consequence, memory leaks and certain bugs, such as dangling pointers and double free bugs, are reduced. Similarly, automatic-signal mechanism consumes computing resources in deciding which thread to be signaled; programmers avoid explicitly invoking *signal* calls. As a result, some bugs, such as using wrong notification and signaling a wrong condition variable, are eliminated. Secondly, in explicit-signal monitor, the principle of separation of concerns is violated. Any method that changes the state of the monitor must be aware of all the conditions, which other threads could be waiting for, in other methods of the monitor. The intricate relation between threads for conditional synchronization breaks the modularity and encapsulation of programming. Finally, *Explicit-Signal*

```
1 class BoundedBuffer {
2   Object[] buff;
3   int putPtr, takePtr, count;
4   Lock mutex = new ReentrantLock();
5   Condition insufficientSpace = mutex.newCondition();
6   Condition insufficientItem = mutex.newCondition();
7   public BoundedBuffer(int n) {
8     buff = new Object[n];
9     putPtr = takePtr = count = 0;
10  }
11  public void put(Object[] items) {
12    mutex.lock();
13    while (items.length + count > buff.length) {
14      insufficientSpace.await();
15    }
16    for (int i = 0; i < items.length; i++) {
17      buff[putPtr++] = items[i];
18      putPtr %= buff.length;
19    }
20    count += items.length;
21    insufficientItem.signalAll();
22    mutex.unlock();
23  }
24  public Object[] take(int num) {
25    mutex.lock();
26    while (count < num) {
27      insufficientItem.await();
28    }
29    Object[] ret = new Object[num];
30    for (int i = 0; i < num; i++) {
31      ret[i] = buff[takePtr++];
32      takePtr %= buff.length;
33    }
34    count -= num;
35    insufficientSpace.signalAll();
36    mutex.unlock();
37    return ret;
38  }
39 }
```

Automatic-Signal

```
1 AutoSynch class BoundedBuffer {
2   Object[] buff;
3   int putPtr, takePtr, count;
4   public BoundedBuffer(int n) {
5     buff = new Object[n];
6     putPtr = takePtr = count = 0;
7   }
8   public void put(Object[] items) {
9     waituntil(count + items.length <= buff.length);
10    for (int i = 0; i < items.length; i++) {
11      buff[putPtr++] = items[i];
12      putPtr %= buff.length;
13    }
14    count += items.length;
15  }
16  public Object[] take(int num) {
17    waituntil(count >= num);
18    Object[] ret = new Object[num];
19    for (int i = 0; i < num; i++) {
20      ret[i] = buff[takePtr++];
21      takePtr %= buff.length;
22    }
23    count -= num;
24    return ret;
25  }
26 }
```

Figure 1: The parameterized bounded-buffer example

AutoSynch can provide rapid prototyping in developing programs and accelerating product time to market.

Although this paper focuses on Java, our techniques are also applicable to other programming languages and models, such as pthread and C# [13].

This paper is organized as follows. Section 2 gives the background of the monitor and explains why *signalAll* is required for explicit-signal monitor but not automatic-signal monitor. The concepts of *AutoSynch* are presented in Section 3 and the practical implementation details are discussed in Section 4. The proposed methods are then evaluated with experiments in Section 5. Section 6 gives the concluding remarks.

2. Background: monitor

According to Buhr and Harji [5], monitors can be divided into two categories according to the different implementations of conditional synchronization.

Explicit-signal monitor In this type of monitor, condition variables, *signal* and *await* statements are used for synchronization. Programmers need to associate assertions with condition variables manually. A thread waits on some condition variable if its predicate is not true. When another thread detects that the state has changed and the predicate is true, it explicitly signals the appropriate condition variable.

Automatic-signal (implicit-signal) monitor This kind of monitor uses *waituntil* statements, such as line 9 in automatic-signal program in Fig. 1, instead of condition variables for synchronization. Programmers do not need to associate assertions with variables, but use *waituntil* statements directly. In monitor, a thread will wait as long as the condition of a *waituntil* statement is false, and execute the remaining tasks only after the condition becomes true. The responsibility of signaling a waiting thread is that of the system rather than of the programmers.

We note that the *signalAll* call is essential in explicit-signal mechanism when programmers do not know which thread should be signaled. In Fig. 1, a producer must wait if there is no space to put num items, while a consumer has to wait when the buffer has insufficient items. Since producers and consumers can put and take different numbers of items every time, they may wait on different conditions to be met. Programmers do not know which producer or consumer should be signaled at runtime. Therefore, the *signalAll* call is used instead of *signal* calls in line 21 and 35. Although programmers can avoid using *signalAll* calls by writing complicated code that associates different conditions to different condition variables; the complicated code makes the maintenance of the program hard.

The *signalAll* call is expensive; it generally decreases the performance because it introduces redundant context switches, requiring computing time to save and load registers and update various tables and lists. Furthermore, *signalAll* calls cannot increase parallelism because threads are forbidden to access a monitor simultaneously. Although multiple threads are signaled at a time, only one thread is able to acquire the monitor. Other threads may need to go back to waiting state since another thread may change the status of the monitor.

3. *AutoSynch* concepts

3.1 Predicate evaluation

In *AutoSynch*, it is the responsibility of the system to signal appropriate threads automatically. The predicate evaluation is crucial in deciding which thread should be signaled. We discuss how to preform predicate evaluations of *waituntil* statements.

A predicate $P(\vec{x}) : X \to \mathbb{B}$ is a Boolean condition, where X is the space spanned by the variables $\vec{x} = (x_1, \ldots, x_n)$. A variable of a monitor object is a shared variable if it is accessible by every thread that is accessing the monitor. The set of shared variables is denoted by S. The set of local variables, denoted by L, is accessible only by a thread calling a function in which the variables are declared.

Predicates can be used to describe the properties of conditions. In our approach, every condition of *waituntil* statement is represented by a predicate. We say a condition has been met if its representing predicate is true; otherwise, the predicate is false. Furthermore, we assume that every predicate, $P = \vee_{i=1}^{n} c_i$, is in disjunctive normal form (DNF), where c_i is defined as the conjunction of a set of atomic Boolean expressions. For example, a predicate $(x = 1) \wedge (y = 6) \vee (z \neq 8)$ is in DNF, where $c_1 = (x = 1) \wedge (y = 6)$ and $c_2 = (z \neq 8)$. Note that, every Boolean formula can be converted into DNF using De Morgan's laws and distributive law.

Predicates can be divided into two categories based on the type of their variables [5].

Definition 1 (Shared and complex predicate)**.** *Consider a predicate $P(\vec{x}) : X \to \mathbb{B}$. If $X \subseteq S$, P is a shared predicate. Otherwise, it is a complex predicate.*

The automatic-signal monitor has an efficient implementation [16] by limiting the predicate of a *waituntil* to a shared predicate; however, we do not limit the predicate of a *waituntil* statement to a shared predicate. The reason is that this limitation will lead *AutoSynch* to be less attractive and practical since conditions including local variables cannot be represented in *AutoSynch*.

Evaluating a complex predicate in all the threads is unattainable because the accessibility of the local variables in the predicate is limited to the thread declaring them. To evaluate a complex predicate in all the threads, we treat local variables as constant values at runtime and define closure as follows.

Definition 2 (Closure)**.** *Given a complex predicate $P(\vec{x}, \vec{a}) : X \times A \to \mathbb{B}$, where $X \subseteq S$ and $A \subseteq L$. The closure of P at runtime **t** is the new shared predicate*

$$G_t(\vec{x}) = P(\vec{x}, \vec{a_t}),$$

*where $\vec{a_t}$ is the values of $\vec{a}$ at runtime **t**.*

The closure can be applied to any complex predicate; a shared predicate can be derived from the closure. For example, in Fig. 1, the consumer C wants to take 48 items at some instant of time. Applying the closure to the complex predicate $(count \geq num)$ in line 19, we derive the shared predicate $(count \geq 48)$.

The following proposition shows that the complex predicate evaluation of *waituntil* statement in all threads can be achieved through the closure.

Proposition 1. *Consider a complex predicate $P(\vec{x}, \vec{a})$ in a **waituntil** statement. $P(\vec{x}, \vec{a})$ and its closure $P(\vec{x}, \vec{a_t})$ are semantically equivalent during the **waituntil** period, where t is the time instant immediately before invoking the **waituntil** statement.*

Proof. Only the thread invoking the *waituntil* statement can access the local variables of the predicate; all other threads are unable to change the values of those local variables. Therefore, the value of $\vec{a}$ cannot be changed during the *waituntil* period. Since $\vec{a_t}$ is the value of $\vec{a}$ immediately before invoking the *waituntil* statement, $P(\vec{x}, \vec{a})$ and $P(\vec{x}, \vec{a_t})$ are semantic equivalent during the *waituntil* period. ■

Proposition 1 enables the complex predicate evaluation of *waituntil* statement in all threads. Given a complex predicate in a *waituntil* statement, in the sequel we substitute all the local variables with their values immediately before invoking the statement. The predicate can now be evaluated in all other threads during the *waituntil* period.

3.2 Relay invariance

As mentioned in Section 2, *signalAll* calls are sometimes unavoidable in the explicit-signal mechanism. In *AutoSynch*, *signalAll* calls are avoided by providing the *relay invariance*.

Definition 3 (Active and inactive thread)**.** *Consider a thread that tries to access a monitor. If it is not waiting in a* waituntil *statement or has been signaled, then it is an* active *thread for the monitor. Otherwise, it is an* inactive *thread.*

Definition 4 (Relay invariance)**.** *If there is a thread waiting for a predicate that is true, then there is at least one active thread; i.e., suppose W_T is the set of waiting threads whose conditions have become true, A_T is the set of active threads, then*

$$W_T \neq \phi \Rightarrow A_T \neq \phi$$

holds at all time.

AutoSynch uses the following mechanism for signaling.

Relay signaling rule: When a thread exits a monitor or goes into waiting state, it checks whether there is some thread waiting on a condition that has become true. If at least one such waiting thread exists, it signals that thread.

Proposition 2. *The relay signaling rule guarantees relay invariance.*

Proof. Suppose a thread T is waiting on the predicate P that is true. Since T is waiting on P, P must be false before T went to waiting state. There must exist another active thread R after T such that R changed the state of the monitor and made P true. According to the rule, R must signal T or another thread waiting for a condition that is true before leaving the monitor or going into waiting state. The thread signaled by R then becomes active. Therefore, the relay invariance holds. ■

Our framework guarantees progress by providing *relay invariance*. The concept behind relay invariance is that, the privilege to enter the monitor is transmitted from one thread to another thread whose condition has become true. For example, in Fig. 1, the consumer C tries to take 32 items; however, only 24 items are in the buffer at this moment. Then, C waits for the predicate $P : (count \geq 32)$ to be true. A producer, D, becomes active after C; D puts 16 items into the buffer and then leaves the monitor. Before leaving, D finds that P is true and then signals C; therefore, C becomes active again and takes 32 items of the buffer. Proposition 2 shows that the relay invariance holds in our automatic-signaling mechanism. Thus, *signalAll* calls are avoidable in *AutoSynch*. Note that, although at most one thread is signaled at any time; the signaled thread is not guaranteed to get the lock. Some other thread trying to acquire the lock could also get the lock. The signaled thread may need to go back as a waiting thread, since the state of the monitor may be changed. However, this situation is very rare in comparison with the *signalAll* call. The problem is now reduced to finding a thread waiting for a condition that is true.

3.3 Predicate tag

In order to efficiently find an appropriate thread waiting for a predicate that is true, we analyze every waiting condition and assign different tags to every predicate according to its semantics. These tags help us prune predicates that are not true by examining the state of the monitor. The idea behind the predicate tag is that, local variables cannot be changed during the *waituntil* period; thus the values of local variables are used as keys when we evaluate predicates. First, we define two types of predicates according to their semantics.

Definition 5 (Local and shared expression)**.** *Consider an expression $E(\vec{x}) : X \to \mathbb{D}$, where $\mathbb{D}$ represents one of the primitive data types in Java. If $X \subseteq L$, then E is a local expression. Otherwise, if $X \subseteq S$, E is a shared expression.*

We use SE to denote a shared expression, and LE to denote a local expression.

Definition 6 (Equivalence predicate)**.** *A predicate $P : (SE = LE)$ is an equivalence predicate.*

Definition 7 (Threshold predicate)**.** *A predicate $P : (SE\ \boldsymbol{op}\ LE)$ is a threshold predicate, where $\boldsymbol{op} \in \{<, \leq, >, \geq\}$.*

Note that, many predicates that are not equivalence or threshold predicates can be transformed into them. Consider the predicate $(x - a = y + b)$, where $x, y \in S$ and $a, b \in L$. This predicate is equivalent to $(x - y = a + b)$ which is an equivalence predicate. Thus, these two types of predicates can represent a wide range of conditions in synchronization problems.

Given an *Equivalence* or a *Threshold* predicate, we can apply the *closure* operation to derive a constant value on the right hand side of the predicate. In *AutoSynch*, there are three types of tags, $Equivalence$, $Threshold$, and *None*. Every $Equivalence$ or $Threshold$ tag represents an equivalence predicate or a threshold predicate, respectively. If the predicate is neither equivalence nor threshold, it acquires the *None* tag. For example, consider the $Threshold$ predicate $x + b > 2y + a$ where a and b are local variables with values 11 and 2. We first use the closure to convert it to $(x - 2y > 9)$, which is represented by the tag $(Threshold,\ x - 2y,\ 9,\ >)$. The formal definition of tag is as follows.

Definition 8. *A tag is a four-tuple $(M,\ expr,\ key,\ op)$, where*

- *$M \in \{Equivalence,\ Threshold,\ None\}$;*
- *expr is a shared expression if $M \in \{Equivalence,\ Threshold\}$; otherwise, $expr = \perp$;*
- *key is the value of a local expression after applying closure if $M \in \{Equivalence,\ Threshold\}$; otherwise, $key = \perp$;*
- *$op \in \{<, \leq, >, \geq\}$ if $M = Threshold$; otherwise, $op = \perp$.*

We say that a tag is true (false) if the predicate representing the tag is true (false).

3.3.1 Predicate tagging

Tags are given to every predicate by the algorithm shown in Fig. 3. A tag is assigned to every conjunction. The tags of conjunctions of a predicate constitute the set of tags of the predicate. When assigning a tag to a conjunction, the equivalence tag has the highest priority because the set of values that make an equivalence predicate true is smaller than the set of values that make a threshold predicate true. For example, the equivalence predicate $x = 8$ is true only when the value of x is 8, whereas the threshold predicate $x > 3$ is true for a much larger set of values. Therefore, the $Equivalence$ tags can help us prune predicates that are false more efficiently than other kinds of tags. If a conjunction does not include any equivalence predicate, then we check whether it includes any threshold predicate. If yes, then a $Threshold$ tag is assigned to the conjunction; otherwise, the conjunction has a *None* tag.

```
tags = empty
foreach conjunction c
  if c contains an equivalence predicate se=le
    tag t = (Equivalence, se, closure(le), null)
  else if c contains a threshold predicate se op le
    tag t = (Threshold, se, closure(le), op)
  else
    tag t = (None, null, null, null)
  add t to tags
return tags
```

Figure 3: Predicate Tagging

Creating all tags for a conjunction is unnecessary. If a conjunction includes multiple equivalence predicates or threshold predicates, only one arbitrary $Equivalence$ tag or $Threshold$ tag is assigned to the conjunction. If there are a large number of tags, then the performance may decrease because of the cost of maintaining tags. Assigning multiple tags to a conjunction cannot accelerate the searching process. For example, consider a conjunction $(x = 8) \wedge (y = 9)$. If only a tag $(Equivalence,\ x,\ 8,\ null)$ is assigned to the conjunction, we check the predicate when the tag is true. Adding another tag $(Equivalence,\ y,\ 9,\ null)$ cannot accelerate the searching process since we need to check both the tags. Note that multiple predicates with a shared conjunct may share a tag. For example, the predicates $(x = 5) \wedge (z \leq 4)$ and $(x = 5) \wedge (y \geq 4)$ would have a shared equivalence tag of $(x = 5)$.

As another example, consider the predicate $p = ((x < 5) \wedge (y = 3)) \vee ((x > 5) \wedge (foo2())) \vee foo1()$, where x and y are shared variables, and, $foo1()$ and $foo2()$ are boolean functions. The predicate p has three tags, $(Equivalence,\ y,\ 3,\ null)$ for the clause $(x < 5) \wedge (y = 3)$, $(Threshold,\ x, 5, >)$ for the clause $(x > 5) \wedge (foo2())$, and *None* tag for $foo1()$.

3.3.2 Tag signaling

Signaling mechanism is based on tags in *AutoSynch*. Since the equivalence tag is more efficient in pruning the search space than the threshold tag, the predicates with equivalence are checked prior to the predicates with other tags. If no true predicate is found after checking *Equivalence* tags and *Threshold* tags, our algorithm does the exhaustive search for the predicates with a *None* tag.

Equivalence tag signaling: Observe that, an equivalence predicate becomes true only when its shared expression equals the specific value of its local expression after applying *closure*. For distinct equivalence tags related to the same shared expression, at most one tag can be true at a time because the value of its local expression is deterministic and unique at any time. By observing the value of its local expression, the appropriate tag can be identified. For example, suppose there are three *Equivalence* tags for predicates $x = 3$, $x = 6$, and $x = 8$. We examine x and find that its value is 8. Then we know that only the third predicate $x = 8$ is true. Based on this observation, for each unique shared expression of an equivalence tag, we create a hash table, where the value of the local expression is used as the key. By using this hash table and evaluating the shared expression at runtime, we can find a tag that is true in $O(1)$ time if there is any. Then we check the predicates having the tag.

Threshold tag signaling: Threshold tag signaling exploits monotonicity of the predicate to reduce the complexity of evaluating predicates. For example, suppose there are two predicates, $x > 5$ and $x > 3$. We know that if $x > 3$ is false, then $x > 5$ cannot be true. Hence, we only need to check the predicate with the smallest local expression value for $>$ and $\geq$ operations. Similarly, the predicate $x > 3$ cannot be true when $x \geq 3$ is false; i.e., we only need to check the predicate $x \geq 3$. We use a min-heap data structure for storing the threshold tags related to a same shared expression with $\boldsymbol{op} \in \{>, \geq\}$. If two predicates have the same local expression value but different operations, then the predicate with $\geq$ is considered to have a smaller value than the predicate with $>$ in the min-heap. Dually, the max-heap is used for threshold tags with $\boldsymbol{op} \in \{<, \leq\}$.

The signaling mechanism for $Threshold$ tag is shown if Fig. 4. In general, the tag in the root of a heap is checked. If the tag is false, all the descendant nodes are also false. Otherwise, all predicates having the tag need to be checked for finding a true predicate. To maintain the correctness, if no predicate is true, the tag is removed from the heap temporarily. Then the tag in the position of the new heap root is checked again until a true predicate is found or a false tag is found. Those tags removed temporarily are reinserted in the heap. The reason to remove the tags is that the descendants of the tags may also be true since the tags are true. So we also need to check the descendant tags. For example, consider the predicates $P_1 : (x \geq 5) \wedge (y \neq 1)$ and $P_2 : (x > 7)$. P_1 has the tag $Q_1 : (Threshold,\ x,\ 5,\ \geq)$ and P_2 has the tag $Q_2 : (Threshold,\ x,\ 7,\ >)$. Q_1 is the root and Q_2 is its descendant. Suppose at some time instant $x = 3$, then Q_1 is false; thus, there is no need to check Q_2. Now, suppose $x = 9$ and $y = 1$, then Q_1 is true. We check all predicates that have tag Q_1. Since P_1 is false, no predicate having tag Q_1 is true. Then Q_1 is removed form the heap temporarily. We find the new root Q_2 is true and P_2 that has tag Q_2 is also true. We signal a thread waiting for P_2 and then add Q_1 back to the heap.

```
list backup = empty;
tag t = heap.peek(); //retrieve but not remove the root
while t is true
  foreach predicate p with t
    if p is true
      signal a thread waiting on p
      foreach b in backup
        heap.add(b)
      return
  backup.insert(heap.poll()) //retrieve and remove the root
  t = heap.peek()
foreach b in backup
  heap.add(b)
```

Figure 4: Threshold tag signaling

Suppose there are n $Threshold$ tags for a shared expression with different keys, and these tags are assigned to m predicates. The time complexity for maintaining the heap is $O(n \log(n))$ However, the performance is generally much better because we only need to check the predicates of the tags in the root position in the most cases. The time complexity for finding the root is $O(1)$. In the worst case, we need to check all predicates; thus, the time complexity is $O(n \log(n) + m)$. However, this situation is rare. Furthermore, this algorithm is optimized for evaluating threshold predicates by sacrificing performance in tag management.

4. *AutoSynch* implementation

The *AutoSynch* implementation involves two parts, the preprocessor and the Java library of condition manager. The preprocessor, built using JavaCC [17], translates *AutoSynch* code to Java code. Our signal-mechanism is implemented in the condition manager library that creates condition variables, and maintains the association between predicates and condition variables. Furthermore, predicate tags are also maintained by the condition manager. It is the responsibility of the condition manager to decide which thread should be signaled.

4.1 Preprocessor

The *AutoSynch* class provides both mutual exclusion and conditional synchronization. To maintain these two properties, our preprocessor adds some additional variables for any *AutoSynch* class. Fig. 5 summarizes the definitions of additional variables in the constructor of an *AutoSynch* class. The lock variable, $mutex$, is declared for mutual exclusion, which is acquired at the beginning of every member function and released before the return statement. In addition, a condition manager, $condMgr$, is declared for synchronization. The details of the condition manager are discussed in the next section.

```
Lock mutex
ConditionManager condMgr
foreach shared predicate P
  tags = AnalyzePredicate(toDNF(P))
  condMgr.registerSharedPredicate(P, tags)
foreach shared expression E
  condMgr.registerSharedExpression(E)
```

Figure 5: The additional variables for an *AutoSynch* class

All predicates are transformed to DNF in the preprocessing process by De Morgan's laws and distributive law. Then predicates are analyzed to derive tags by our preprocessor. The shared predicates and shared expressions are identified in the preprocessing stage and registered to the condition manager in the constructor of the class for predicate evaluation at runtime as in Fig. 5. Shared predicates and shared expressions (but not complex predicates) are registered in the construct because their semantics is static and never changes. A complex predicate is registered dynamically because its closure may change according to the value of its local variables at runtime. In Java, the predicates and shared expressions are created as inner classes that can access the shared variables appearing in them with $isTrue()$ or $getValue()$ functions for the condition manager to evaluate. The function $isTrue()$ returns the evaluation of a predicate and the function $getValue()$ returns the value of a shared expression.

For every member function of an *AutoSynch* class, the *mutex.lock()* and *mutex.unlock()* are inserted at the beginning of the function and immediately before the *return* statement, respectively.

In the *waituntil* statement, the predicate is checked initially at runtime. If it is true, then the thread can continue. Otherwise, the type of predicate is checked. If the predicate is complex, then $closure$ is applied to it for deriving a new shared predicate. Then the condition manager is queried to determine whether the derived predicate has been registered earlier. If not, the condition manager registers the predicate with its tags. The corresponding condition variable can be obtained by calling *getCondition()* function of the condition manager. The *relaySignal()* function is called to maintain relay invariance and signal an appropriate thread. Then, the thread goes into the waiting state until its predicate becomes true and some other thread signals it. After exiting the waiting state, if the predicate is complex and the corresponding condition has no other waiting thread, the predicate is deactivated by the condition manager.

```
if P is false
  if P is a complex predicate
    P := Closure(toDNF(P))
    if P is not in condMgr
      tags = AnalyzePredicate(P)
      condMgr.registerComplexPredicate(P, tags)
  C = condMgr.getCondition(P)
  do
    condMgr.relaySignal()
    wait C
  while P is false
  if P is complex predicate and C has no waiting thread
    condMgr.deactivate(P)
```

Figure 6: Preprocessing for a *waituntil(P)* statement

4.2 *AutoSynch* Java library: condition manager

The condition manager maintains the predicates and condition variables, and provides the signaling mechanism. To avoid creating redundant predicates and condition variables, predicates that have the same meaning should be mapped to the same condition variable. Two predicates are syntax equivalent if they are identical after applying closure. A predicate table, which is implemented by using a hash table, records predicates and their associated condition variables.

When a predicate is added to the condition manager, its tags are stored in an appropriate data structure depending upon the type of its tag. Fig. 7 shows an example. The symbol • indicates a condition variable. The gray blank indicates that the predicate is deactivated, that is, no thread waits on it. A hash table is used for storing equivalence tags with the shared expression x. In addition, a min-heap and a max-heap are used for storing threshold tags.

For finding a predicate that is true in Fig. 7, the value of the shared expression x is evaluated. The hash table of the equivalence tag is checked (with $O(1)$ time complexity) using the value of the shared expression as the key. If a tag is found, then the predicates that have the tag are evaluated. If there exists a predicate that is true, then its corresponding condition variable is signaled. Otherwise, the max-heap and the min-heap of the threshold tag are checked. If both tags of their roots are false, the predicates with the *None* tag are searched exhaustively. If one of these predicates is true, its corresponding condition variable is signaled. As can be expected, the equivalence and threshold tags are helpful for searching predicates that are true.

A predicate must be removed from the predicate table once it has become deactivated to avoid unnecessary predicate evaluation. A threshold tag also needs to be removed once all of its predicates have become deactivated.

Predicates may be reused. Instead of removing deactivated predicates, those predicates are moved to a deactivated list. If they are used later, then we remove them from the deactivated list and add them back to the predicate table and its tags. Otherwise, when the length of the deactivated list exceeds some predefined threshold, we remove the oldest predicates from the list. Note that, the shared predicates are never removed since they are static and are added only at the constructor level.

5. Evaluation

We present the experimental setup and its results to evaluate the performance of *AutoSynch* in this section. We compare the performance of different signaling mechanisms in three sets of classical conditional synchronization problems. The first set of problems relies on only shared predicates for synchronization. Next, we explore the performance for problems using complex predicates. Finally, we evaluate the problems in which *signalAll* calls are required in the explicit-signal mechanism.

5.1 Experimental environment

All of the experiments were conducted on a machine with 16 Intel(R) Xeon(R) X5560 Quad Core CPUs (2.80 GHz) and 64 GBs memory running Linux 2.6.18.

We conducted two types of experiments. The first is a *saturation* test [5], in which only monitor accessing functions performed. That is, no extra work is performed in the monitor or out of the monitor. The other set of experiments simulate different workloads of the monitors [4]. For each monitor operation, there is a fixed time to perform other operations out of the monitor. For every experiment, we ran the program 25 times, and removed the best and the worst results. Then we compared the average runtime for different signaling mechanisms.

We do not report memory usage due to space limitations. Although some additional data structures are created in our framework, the additional memory consumption is insignificant. The reason is that, whenever a predicate has no waiting thread, it is put in an inactive list for reuse. If the size of the inactive list exceeds a predefined threshold, e.g. $2n$ (n is the number of threads), then we remove the oldest predicate and the conditional variable from the list and the table. Moreover, the size of active predicates is always less than n.

5.2 Signaling mechanisms

Four implementations using different signaling mechanisms have been compared.

Explicit-signal Using the original Java explicit-signal mechanism.

Baseline Using the automatic-signal mechanism relying on only one condition variable. It calls *signalAll* to wake every waiting thread. Then each thread that wakes up re-evaluates its own predicate after re-acquiring the monitor.

AutoSynch-T Using the approach described in this paper but excluding predicate tagging.

AutoSynch Using the approach described in this paper.

5.3 Test problems

Seven conditional synchronization problems are implemented for evaluating our approach.

5.3.1 Shared predicate synchronization problems

Bounded-buffer [8, 10] This is the traditional bounded-buffer problem. Every producer waits if the buffer is full, while every consumer waits if the buffer is empty.

H_2O **problem [1]** This is the simulation of water generation. Every H atom waits if there is no O atom or another H atom. Every O atom waits if the number of H atoms is less than 2.

5.3.2 Complex predicate synchronization problems

Round-Robin Access Pattern Every test thread accesses the monitor in round-robin order.

Readers/Writers [7] We use the approach given in [5], where a ticket is used to maintain the accessing order of readers and

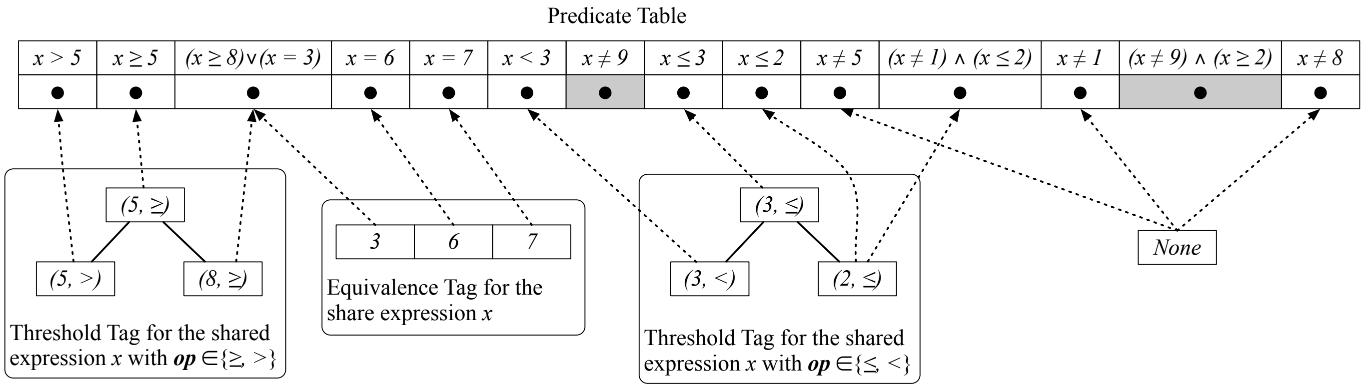

Figure 7: A example of the condition manager in *AutoSynch*

writers. Every reader and writer gets a ticket number indicating its arrival order. Readers and writers wait on the monitor for their turn.

Dining philosophers [10] This problem requires coordination among philosophers sitting around a table and is described in [10].

5.3.3 Synchronization problems requiring ***signalAll*** in explicit

Parameterized bounded-buffer [8, 10] The parameterized bounded-buffer problem shown in Fig. 1.

5.4 Experimental results

Fig. 8 and 9 plot the results for the bounded-buffer and the H_2O problem. The y-axis shows the runtime in seconds. The x-axis represents the number of simulating threads. Note that, in the H_2O problem, only one thread simulates an O atom. The x-axis represents the number of threads simulating H atoms. As expected, the baseline is much slower than other three signaling mechanisms, which have similar performance in the both problems. This phenomenon can be explained as follows. There is only a constant number of shared predicates in *waituntil* statements for automatic-signal mechanisms. For example, in the bounded-buffer problem, there are two *waituntil* statements with global predicates, *count* > 0 (not empty condition) and *count* $<$ *buff.length* (not full condition). Therefore, the complexity for signaling a thread in *AutoSynch* and *AutoSynch-T* is also constant. Hence, both *AutoSynch* and *AutoSynch-T* are as efficient as the explicit-signal mechanism. These experiments illustrate that the automatic-signal mechanisms are as efficient as the explicit-signal mechanisms for synchronization problems relying on only shared predicates.

Fig. 10, 11 12 present the experimental results for the round-robin access pattern, the readers/writers problem, and the dining philosophers problem. The result of the baseline is not plotted in these figures since its performance is extremely inefficient in comparison with other mechanisms. In this set of experiments, the explicit-signal mechanism has an advantage since it can explicitly signal the next thread to enter the monitor. For example, in the round-robin access pattern, an array of condition variables is used for associating the id of each thread and its condition variable. Each thread waits on its condition variable until its turn. When a thread leaves the monitor, it signals the condition variable of the next thread. As can be seen, the performance of explicit-signal mechanism is steady as the number of threads increases in the round-robin access pattern and the reader/writers problem. In *AutoSynch-T*, its runtime increases significantly as the number of threads increase. For *AutoSynch*, the performance is between 1.2 to 2.6 times slower

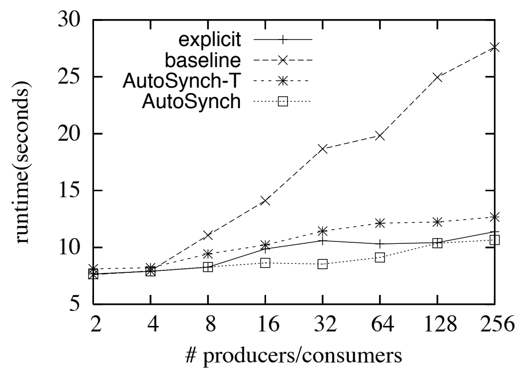

Figure 8: The results of bounded-buffer problem

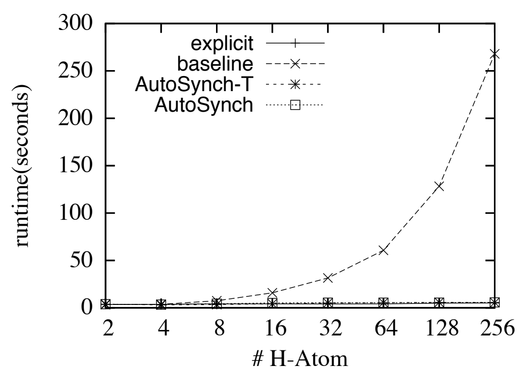

Figure 9: The results of H_2O problem

than the explicit-signal mechanism for the round-robin access pattern. However, the performance of *AutoSynch* does not decrease as the number of threads increases. Note that, in the readers/writers problem, the *AutoSynch-T* is more efficient than *AutoSynch* when the number of threads is small. The reason is that *AutoSynch* sacrifices performance for maintaining predicate tags. The benefit of predicate tagging increases as the number of threads increases. Another interesting point is that the performance of the explicit signal mechanism does not outperform implicit signal mechanisms much in the dining philosophers problem. The reason is that a philosopher only competes with two other philosophers sitting near him even when the number of philosophers increases.

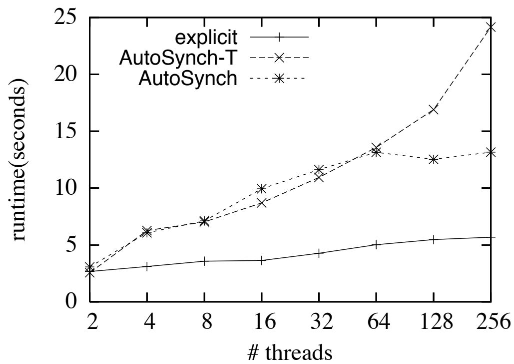

Figure 10: The results of round-robin access pattern

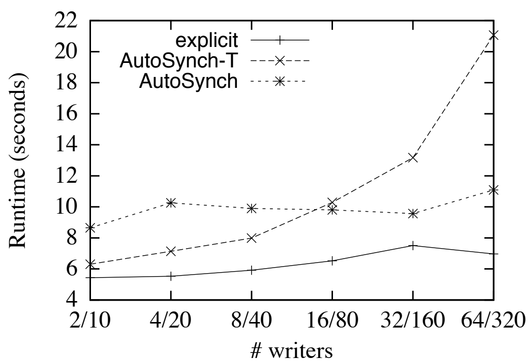

Figure 11: The results of readers/writers problem

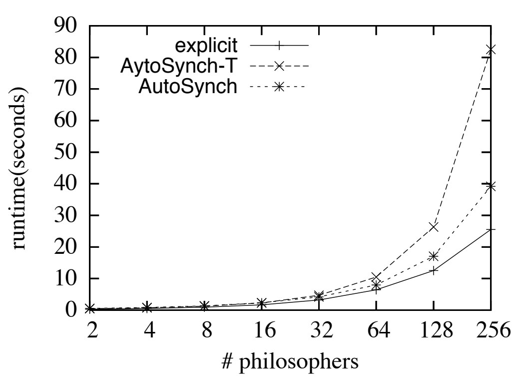

Figure 12: The results of dining philosophers problem

Table 1 presents the CPU usage (profiled by YourKit [20]) for the round-robin access pattern with 128 threads. The *relaySignal* is the process of deciding which thread should be signaled in both *AutoSynch* and *AutoSynch-T*. *Tag Mger* is the computation for maintaining predicate tags in *AutoSynch*. As can be seen, the predicate tagging significantly improves the process for finding a predicate that is true. The CPU time of *relaySingal* process is reduced 95% with a slightly increased cost in tag management.

In Fig. 13, we compare the results of the parameterized bounded-buffer in which *signalAll* calls are required in the explicit-signal mechanism. In this experiment, there is one producer, which randomly puts 1 to 128 items every time. The y-axis indicates the number of consumers. Every consumer randomly takes 1 to 128 items every time. As can be seen, the performance of the explicit-signal mechanism decreases as the number of consumers increases. *AutoSynch* outperforms the explicit-signal mechanism by 26.9 times when the number of threads is 256. This can be explained by Fig. 14 that depicts the number of contexts switches. The number of context switches increases in the explicit-signal mechanism in which the number of context switches is around 2.7 million when the number of threads is 256. However, the numbers of context switches are stable in *AutoSynch* even when the number of threads increase. It has around 5440 context switches when the number of threads is 256. This experiment demonstrates that the number of context switches can be dramatically reduced and the performance can be increased in *AutoSynch* for the problems requiring *signalAll* calls in the explicit-signal mechanism.

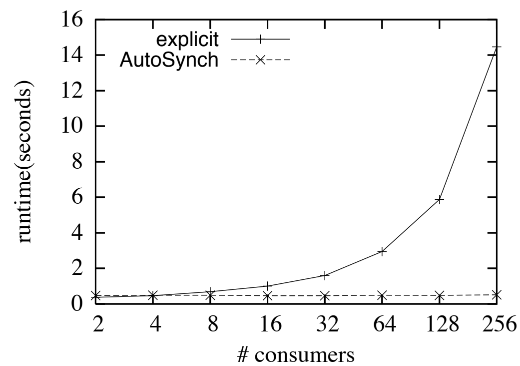

Figure 13: The results of the parameterized bounded-buffer problem

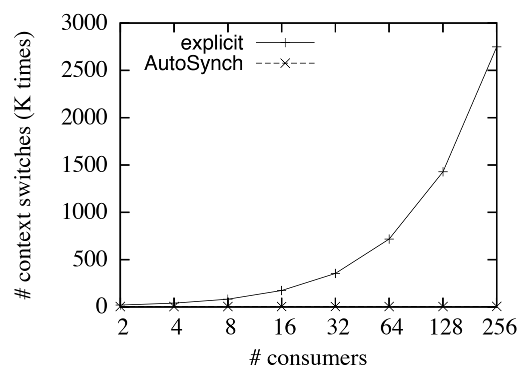

Figure 14: The number of context switches of the parametrized bounded-buffer problem

Fig. 15 and 16 present the run time ratio of our approaches and the *explicit* approach for the round-robin access pattern with 256 threads and the readers/writers problem with 64 writers and 320 readers. The y-axis indicates the runtime ratio and the x-axis shows the delay time, the amount of time in which the threads perform operations out of the monitor between every two monitor operations, in microseconds. As expected, the performance difference decreases as the duration of delay time increases. *AutoSynch* is two times slower than the explicit approach with no delay time (saturation test) for round-robin access pattern. However, when the duration of delay time is 5000 microseconds, *AutoSynch* is only 7.7%

	await		lock		relaySignal		Tag Mger		others		total
	T	%	T	%	T	%	T	%	T	%	T
explicit	21365	99.7%	28	0.15%	NA	NA	NA	NA	28	0.15%	21433
AutoSynch-T	410377	98.5%	3140	0.7%	2108	0.5%	NA	NA	1033	0.2%	416658
AutoSynch	96754	98.8%	812	0.8%	112	0.1%	124	0.1%	148	0.02%	97950

Table 1: The CPU usage for the round robin access pattern

slower than the explicit approach. Note that, even *AutoSynch-T* performs well when the duration of delay time increases. The similar observation can be seen for the readers/writers problem in Fig. 16. The results suggest that our approach could be more useful for practical problems that perform more monitor unrelated operations.

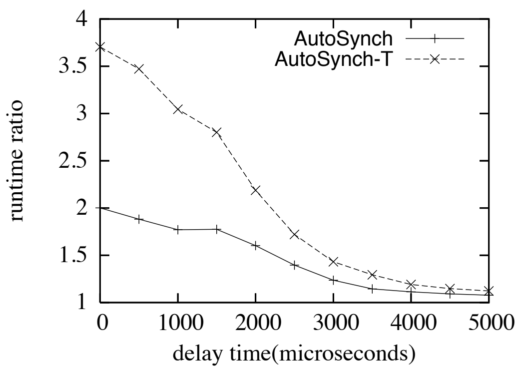

Figure 15: The runtime ratio of round-robin

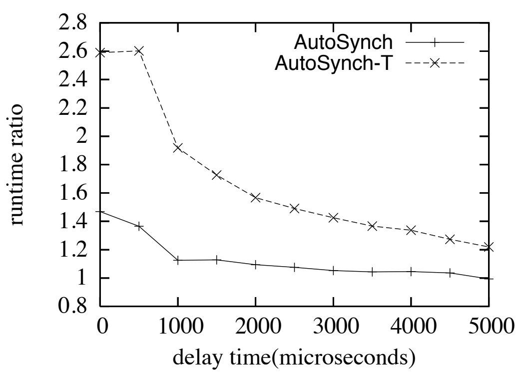

Figure 16: The runtime ratio of ticket readers/writers

6. Future Work

Fairness is not guaranteed in our system. Our approach favors signaling a thread waiting on a predicate that is likely to be true. We focus on the efficiency of our automatic-signal monitor at this point. Fairness may be an important issue for some concurrent applications and we plan to address it in future. Buhr and Harji simulated implicit-signal monitors with explicit monitors [5]. Their approach provides fairness and deals with staleness/freshness issues. However, their approach does not focus on the efficiency.

In the future, we plan to optimize our framework through using the architecture information. For example, we can get the number of cores of a machine, and then limit the number of executing threads to avoid unnecessary contention. Our current implementation of *AutoSynch* is built upon constructs provided by Java. Thus, there is possibility of further performance improvement if the approach was to be implemented within the JVM.

Acknowledgement

The authors thank Himanshu Chauhan, Yen-Jung Chang, John Bridgman, and Craig Chase for the helpful discussions and comments to improve the paper. This work was supported in part by the NSF Grants CNS-1115808, CNS-0718990, and Cullen Trust for Higher Education Endowed Professorship.

References

[1] G. R. Andrews. *Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, MA, 2000.

[2] P. Brinch Hansen, *The Programming language concurrent Pascal*. IEEE Trans. Software Engineering, 1(2), 199–207, June, 1975.

[3] P. Brinch Hansen, *Structured multiprogramming*. Commun. ACM 15(7), 574–578, July 1972.

[4] P. A. Buhr, M. Fortier, and M. H. Coffin, *Monitor Classification*. ACM Computing Surveys, ACM, 27(1), 63–107, Mar. 1995.

[5] P. A. Buhr and A. S. Harji, *Implicit-signal monitors*. ACM Transactions on Programming Languages and Systems ACM, 27(6):1270–1343, Nov. 2005.

[6] D. R. Butenhof, *Programming With Posix Threads*. Addison-Wesley, MA, 1997.

[7] P. J. Courtois, F. Heymans, D. L. Parnars, *Concurrent control with "readers" and "writers"*. Commun. ACM, 14(10), 667–668, Oct. 1971.

[8] E. W. Dijkstra, *Cooperating Sequential Processes*. Technical Report EWD-123, 1965.

[9] E. W. Dijkstra, *The structure of the "THE"-multiprogramming system*. Commun. ACM, 11(5), 341 – 346, May 1968.

[10] E. W. Dijkstra, *Hierarchical Ordering of Sequential Processes*. Acta Informatica, 1, 115 – 138, 1971.

[11] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. 2nd ed. Addison-Wesley, MA, 2000.

[12] V. K. Garg and N. Mittal, *A Critique of Java for Concurrent Programming*, IEEE Distributed Systems online 6(9), 2005.

[13] A. Hejisberg, S. Wiltamuth, and P. Golde, *C# Language Specification*. Addison-Wesley, MA, 2003.

[14] C. A. R. Hoare, *Monitors: an operating system structuring concept*. Commun. ACM, 17(10), 549–557, Oct. 1974.

[15] Y.-J. Joung, *Asynchronous group mutual exclusion*. ACM Symposium on Principles of Distributed Computing, 51–60, Aug. 1998.

[16] J. L. W. Kessels. *An alternative to event queues for synchronization in monitors*. Commun. ACM, 20(7), 500–503, July 1977.

[17] V. Kodaganallur, *Incorporating language processing into Java applications: a JavaCC tutorial*. IEEE Software, 21(4), 70–77, July–Aug. 2004.

[18] D. Lea, *The java.util.concurrent synchronizer framework*. Science of Computer Programming, 58(3), 293 – 309, Dec. 2005.

[19] B. Stroustrup, *The C++ Programming Language*. 3rd ed. Addison-Wesley, MA, 1997.

[20] Yourkit LLC, *Yourkit profiler*. http://www.yourkir.com