УДК 004.272

В. А. Смирнов, А. Р. Омельниченко, А. А. Пазников
Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

Алгоритмы реализации потокобезопасных ассоциативных массивов на основе транзакционной памяти

Предложены алгоритмы реализации потокобезопасных ассоциативных массивов (красно-черное дерево, хеш-таблица с открытой адресацией на основе метода Hopscotch hashing разрешения коллизий) с использованием программной транзакционной памяти (software transactional memory). Представлен анализ эффективности ассоциативных массивов при разном количестве задействованных потоков и процессорных ядер, приведено сравнение с аналогичными структурами данных на основе крупнозернистых и мелкозернистых блокировок, также сформулированы рекомендации по выбору алгоритмов выполнения транзакций. Описаны принципы работы программной транзакционной памяти, различные политики обновления объектов в памяти и стратегии обнаружения конфликтов. Представлены различные методы блокировки при использовании транзакционной памяти, реализованной в компиляторе GCC 5.4.0. Коротко рассмотрены альтернативы, применяемые в настоящее время, выделены их достоинства и недостатки.

Транзакционная память, потокобезопасные структуры данных, красно-черное дерево, хеш-таблица, многопоточное программирование, синхронизация многопоточных программ

Многоядерные вычислительные системы (ВС) с общей памятью в настоящее время используются для решения различных сложных задач. К таким ВС относятся SMP- и NUMA-системы. В связи с ростом количества процессорных ядер остро стоит проблема обеспечения масштабируемого доступа параллельных потоков к разделяемым структурам данных.

Синхронизация доступа к разделяемым ресурсам является одной из важных и сложных задач при разработке многопоточных программ. На сегодняшний день основными методами решения данной проблемы являются:

1. Применение блокировок (mutex, spinlock, critical section и т. д.).

2. Алгоритмы и структуры данных, свободные от блокировок (lockless, lock-free).

3. Транзакционная память (transactional memory).

При использовании традиционных примитивов синхронизации (семафоры, мьютексы, спинлоки и др.) необходимо обеспечить не только корректность программы – отсутствие взаимных блокировок и состояний гонок за данными (data race), – но и минимизировать время ожидания доступа к критическим секциям (разделяемым ресурсам). Классические методы синхронизации, основанные на механизме блокировок, позволяют организовывать в параллельных программах критические секции, выполнение которых возможно только одним потоком в каждый момент времени [1].

При использовании блокировок наблюдается альтернатива использования крупнозернистых и мелкозернистых блокировок. Потокобезопасные структуры на основе крупнозернистых блокировок (coarse-grained lock) легко реализовывать, но они не обеспечивают предельных показателей эффективности, так как обладают ограниченным параллелизмом выполнения операций. Мелкозернистые блокировки (fine-grained lock) обеспечивают хорошую производительность, но их использование – сложная задача [2].

Структуры данных, свободные от блокировок, строятся на базе атомарных операций, таких, как запись (atomic store), чтение (atomic load), сравнение с обменом (compare and swap, CAS) и др. Дан-

ный подход позволяет полностью избавиться от взаимных блокировок, однако может привести к возникновению активных блокировок (livelock) – ситуаций, когда 2 потока одновременно пытаются изменить структуру данных, но каждый из них циклически выполняет свою операцию из-за изменений, произведенных другим потоком. Также к недостаткам данного подхода можно отнести сложность реализации структур, свободных от блокировок [3], и проблему ABA освобождения памяти.

Программная транзакционная память. На сегодняшний день транзакционная память является одним из наиболее перспективных механизмов синхронизации, ее использование позволяет выполнять не конфликтующие между собой операции параллельно. Транзакционная память упрощает параллельное программирование, выделяя группы инструкций в атомарные транзакции – конечные последовательности операций транзакционного чтения/записи памяти [4]. Изменения, вносимые потоком внутри транзакционных секций, незаметны другим потокам до тех пор, пока транзакция не будет зафиксирована (commit). Если во время выполнения транзакции потоки обращаются к одной области памяти и один из потоков совершает операцию записи, то транзакция одного из потоков отменяется (cancel, rollback). При выполнении транзакционной секции одним потоком другие потоки наблюдают состояние либо непосредственно до, либо непосредственно после выполнения транзакции. Важным свойством транзакционной памяти является линеаризуемость выполнения транзакций: ряд успешно завершенных транзакций эквивалентен некоторому последовательному их выполнению. Таким образом, транзакции обладают качествами атомарности, согласованности, изолированности, устойчивости (atomicity, consistency, isolation, durability – ACID) [5]. *Атомарность* – транзакция представляет собой единое целое, не может быть частичной транзакции, другими словами, если выполнена часть транзакции, она отменяется. *Согласованность* – транзакция не нарушает логику и отношения между элементами данных. *Изолированность* – результаты транзакции не зависят от предыдущих или последующих транзакций. *Устойчивость* – после своего завершения транзакция сохраняется в системе, происходит фиксация транзакции.

Для выполнения транзакционных секций runtime-системой компилятора создаются транзакции. Операция транзакционного чтения копирует содержимое указанного участка общей памяти в соответствующий участок локальной памяти потока. Транзакционная запись копирует содержимое указанного участка локальной памяти в соответствующий участок общей памяти, доступной всем потокам [1].

Основными различиями, определяющими эффективность программной транзакционной памяти, являются политика обновления объектов в памяти и стратегия обнаружения конфликтов.

Политика обновления объектов в памяти определяет, когда изменения объектов внутри транзакции будут зафиксированы. Существуют две основные политики – ленивая и ранняя [6]. В случае *ленивой политики* (lazy version management) все операции с объектами откладываются до момента фиксации транзакции. Все операции записываются в специальном журнале изменений (redo log), который при фиксации транзакции используется для отложенного выполнения операций. Использование журнала изменений замедляет операцию фиксации, но существенно упрощает процедуры ее отмены и восстановления.

Ранняя политика обновления объектов в памяти (eager version management) предполагает, что все изменения объектов сразу записываются в память. В журнале отката (undo log) фиксируются все выполненные операции с памятью. Он используется для восстановления оригинального состояния модифицируемых участков памяти в случае возникновения конфликта. Эта политика характеризуется быстрым выполнением операции фиксации транзакции, но медленным выполнением процедуры ее отмены [6].

Момент времени, когда инициируется алгоритм обнаружения конфликта, определяется стратегией обнаружения конфликтов [6]. При отложенной стратегии (lazy conflict detection) процедура обнаружения конфликтов запускается на этапе фиксации транзакции. Недостатком этой стратегии является то, что временной интервал между возникновением конфликта и его обнаружением может быть достаточно большим.

Пессимистичная стратегия обнаружения конфликтов (eager conflict detection) запускает алгоритм их обнаружения при каждой операции обращения к памяти. Такой подход позволяет избежать недостатков отложенной стратегии, но может привести к значительным накладным расходам.

В данной статье описана реализация транзакционной памяти в компиляторе GCC (библиотека libitm), в котором используется ранняя политика

обновления объектов в памяти и реализован комбинированный подход к обнаружению конфликтов – отложенная стратегия используется совместно с пессимистической [1]. Алгоритм выполнения транзакции представлен на рис. 1.

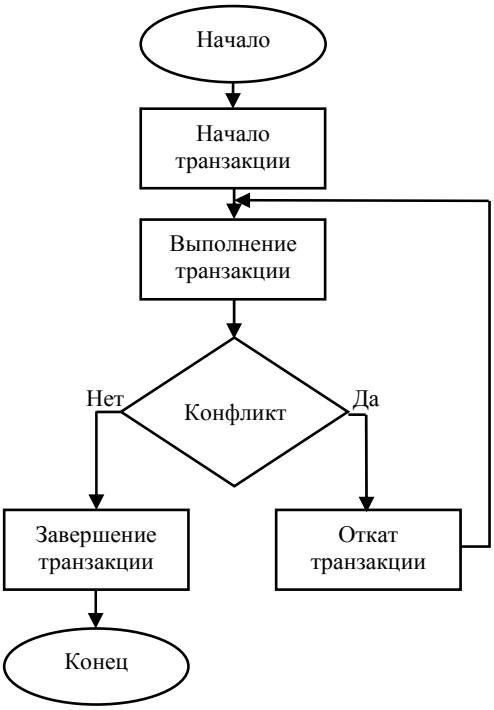

Рис. 1

Для реализации потокобезопасных структур данных используется программная транзакционная память. К одним из наиболее распространенных структур данных в многопоточных программах относятся потокобезопасные ассоциативные массивы.

В настоящее время существует множество реализаций ассоциативных массивов, например: контейнер map в библиотеке STL языка C++, тип Dictionary в C#, класс Hash в Ruby, тип словаря в Python и Java.util.map в Java.

В статье предлагаются алгоритмы реализации потокобезопасных хеш-таблиц (на основе алгоритма Hopscotch hashing разрешения коллизий) и красно-черных деревьев поиска с использованием транзакционной памяти.

Алгоритм хеширования Hopscotch hashing. Хеш-таблица – одна из наиболее широко используемых структур данных при реализации ассоциативных массивов. Повышение ее эффективности позволит сократить время выполнения большого числа параллельных программ [7]. Хеш-таблица хранит пары (ключ, значение) и обеспечивает выполнение операций добавления новой пары, поиска заданного ключа и удаления пары по ключу за константное время [8].

В хеш-таблице с открытой адресацией, в отличие от хеш-таблицы с закрытой адресацией, в ячейке хранится не указатель на связный список, а один элемент (ключ, значение). Если при вставке элемента соответствующая ячейка занята, алгоритм вставки проверяет следующие ячейки до тех пор, пока не будет найдена свободная. Данный подход характеризуется хорошей локальностью кеш-памяти, поскольку каждая кеш-линия содержит сразу несколько записей хеш-таблицы. Существенным недостатком этого подхода является снижение производительности по мере заполнения хеш-таблицы.

Hopscotch hashing [7] объединяет в себе преимущества трех подходов: Cuckoo hashing [9], метода цепочек [10] и метода линейного хеширования [10]. Алгоритм обладает высоким коэффициентом попадания в кеш-память. В худшем случае временная сложность операции добавления – $O(n)$, в лучшем случае – $O(1)$. Операции поиска и удаления выполняются за константное время.

Основная идея Hopscotch hashing заключается в использовании окрестности каждой ячейки массива. Искомый элемент должен находиться в окрестности ячейки, на которую указывает хеш-функция. Таким образом, используется свойство пространственной локальности кеш-памяти, и время поиска элемента в окрестности близко к времени поиска в одной ячейке. Это достигается при вставке элемента вытеснением других элементов.

Пусть задана хеш-функция $h(x)$. Тогда элемент с ключом x всегда может быть найден в ячейке $h(x)$ либо в одной из следующих $H-1$ ячеек, где H – константа, представляющая размер окрестности (в данном случае $H=32$ – стандартный размер машинного слова). Другими словами, окрестность имеет фиксированный размер и пересекается со следующими $H-1$ окрестностями. Каждая ячейка хеш-таблицы включает в себя битовую карту hop размером H бит, которая указывает, какие из следующих $H-1$ ячеек содержат элементы со значением хеш-функции текущей ячейки. Элемент можно найти просмотрев H записей (на большинстве машин это требует не более двух загрузок строк кеша).

Каждая ячейка содержит информацию о том, какие ячейки в окрестности имеют ключ с таким же значением хеш-функции. В данной реализации эта информация представлена в виде связного списка: в

ячейке содержатся относительные позиции следующей и первой ячеек в списке. Таким образом, чтобы найти элемент с ключом x, достаточно проверить ячейку $h(x)$ и ячейки в ее окрестности.

Ниже представлена функция добавления элементов в хеш-таблицу. Критическая секция выделена в транзакцию (строки 4–23). Это позволяет потокам добавлять элементы в хеш-таблицу параллельно. Если элемент с заданным ключом уже находится в хеш-таблице, функция возвращает false (строка 6). Если в пределах *ADD_RANGE* (в данной реализации *ADD_RANGE* = 256) пустую ячейку найти не удалось, происходит возврат false, а функция Resize() изменяет размер хеш-таблицы и производит рехеширование (строки 21, 22). Если элемент успешно добавлен в таблицу, функция возвращает true (строка 19).

```
1: procedure HopscotchInsert
2:   hash = HashFunc(key)
3:   start_bucket = segments_arys + hash
4:   transaction
5:     if Contains(key) then
6:       return false
7:     end if
8:     free_bucket_index = hash
9:     free_bucket = start_bucket
10:    distance = 0
11:    FindFreeBucket(free_bucket, distance)
12:    if distance < ADD_RANGE then
13:      if distance > HOP_RANGE then
14:        FindCloser(free_bucket, distance)
15:      end if
16:      start_bucket.hop_info |= (1 << distance)
17:      free_bucket.data = data
18:      free_bucket.key = key
19:      return true
20:    end if
21:    Resize()
22:    return false
23: end transaction
```

Пример выполнения операции вставки представлен на рис. 2.

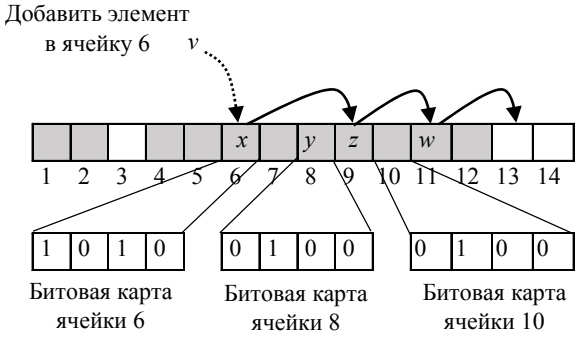

Рис. 2

Ниже представлена функция удаления элемента из хеш-таблицы. Критическая секция функции удаления элемента также выделена в транзакцию (строки 5–20). Если удаление элемента прошло успешно, функция возвращает true (строка 15), в противном случае – false (строка 19). Для обеспечения максимальной производительности был выбран минимально возможный размер транзакционной секции.

```
1: procedure HopscotchRemove
2:   hash = HashFunc(key)
3:   start_bucket = segments_arys + hash
4:   mask = 1
5:   transaction
6:     hop_info = start_bucket.hop_info
7:     for i = 0 to HOP_RANGE do
8:       mask <<= 1
9:       if mask & hop_info then
10:        check_bucket = start_bucket + i
11:      if key = check_bucket.key then
12:        check_bucket.key = NULL
13:        check_bucket.data = NULL
14:        start_bucket.hop_info &= ~(1 << i)
15:        return true
16:      end if
17:    end if
18:  end for
19:  return false
20: end transaction
```

Потокобезопасное красно-черное дерево. Рассмотрим реализацию потокобезопасного красно-черного дерева на основе транзакционной памяти. Красно-черные деревья широко применяются при реализации ассоциативных массивов, обеспечивают логарифмический рост высоты дерева в зависимости от числа узлов и выполняют основные операции дерева поиска: добавление, удаление и поиск узла за $O(\log n)$. Сбалансированность достигается за счет введения дополнительного атрибута узла дерева – «цвета» [8]. Пример красно-черного дерева представлен на рис. 3.

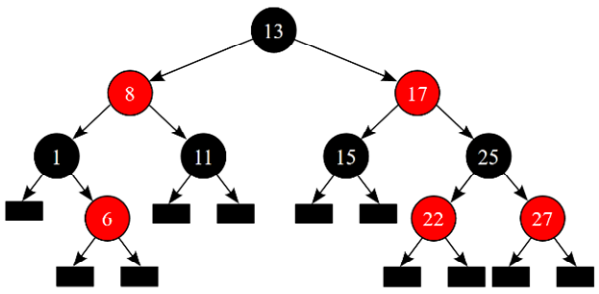

Рис. 3

Ниже представлена функция добавления элемента в красно-черное дерево. Критическая секция выделена в транзакцию (строки 3–10), что позволяет потокам выполнять добавление элементов, не нарушая целостность данных и баланс дерева.

```
1: procedure RBTREEINSERT
2:   x = NEWNODE(data)
3:   transaction
4:     if !FINDPARENT(x) then
5:       return false
6:     end if
7:     INSERTNODE(x)
8:     INSERTBALANCE(x)
9:   return true
10: end transaction
```

Функция INSERTNODE вставки узла (строка 8) определяет поля left или right родительского узла. Если родительский узел отсутствует, добавляемый узел становится корнем дерева. Функция создания нового узла NEWNODE (строка 2) вынесена за пределы транзакционной секции, что позволяет сократить ее размер, а следовательно, уменьшить количество конфликтов между разными транзакциями.

Результаты экспериментов. Эксперименты проводились на вычислительной системе, оборудованной 4-ядерным процессором Intel i5-3470 (отсутствует поддержка аппаратной транзакционной памяти) и объемом кеш-памяти L1 – 32 Кбайт, L2 – 256 Кбайт, L3 – 6 Мбайт. Размер оперативной памяти – 8 Гбайт.

Используемое программное обеспечение: Linux Mint 18.1, компилятор GCC 5.4.0.

Тест представлял собой запуск p потоков, выполняющих операции добавления, удаления и поиска элемента. Вид операции выбирался случайным образом. Количество операций поиска – 40 %, вставки элемента – 30 %, удаления – 30 %. Число потоков p в тестовых программах варьировалось от 2 до 32. На первом этапе моделирование проводилось для $p = 1, \ldots, 4$ потоков, что не превышает количества ядер процессора. На втором этапе количество потоков доходило до 32. В данном эксперименте использовалось 4 метода выполнения транзакций, реализованных в компиляторе GCC:

● Метод глобальной блокировки (gl_wt) – потоки выполняют транзакции параллельно, глобальная блокировка возникает, когда потоки начинают изменять один участок памяти.

● Метод множественной блокировки (ml_wt) – потоки выполняют транзакции параллельно, пока не выполняют запись в один участок памяти; множественная блокировка транзакций возникает, когда потоки выполняют запись в один участок памяти.

● Последовательные методы (serial, serialirr) – в serial все транзакции выполняются последовательно. В serialirr чтение идет параллельно, а при появлении операции записи транзакция переходит в irrevocable режим, предотвращая несанкционированные записи.

Также для экспериментов использовались реализации структур данных (красно-черное дерево и хеш-таблица) на основе крупнозернистых (coarse-grained) и мелкозернистых (fine-grained) блокировок (только для хеш-таблицы).

В качестве показателя эффективности использовалась пропускная способность $b = N/t$, где N – количество выполненных операций, а t – время выполнения всех операций.

Результаты моделирования хеш-таблицы показаны на рис. 4 и 5.

Хеш-таблица на основе транзакционной памяти обеспечивает большую пропускную способность, по сравнению с реализацией на основе крупнозернистых блокировок, при любом количестве потоков (рис. 5). Методы gl_wt и ml_wt демонстрируют рост пропускной способности с увеличением числа потоков и практически не уступают в производительности реализации на основе мелкозернистых блокировок при условии, что число потоков не превышает числа процессорных ядер (рис. 4). Если число потоков больше 16, эффективность всех методов выполнения транзакций сопоставима. При количестве потоков, превышающем количество процессорных ядер, любой из четырех представленных методов выполнения транзакций уступает fine-grained алгоритмам.

На рис. 6 и 7 представлены результаты моделирования потокобезопасного красно-черного дерева.

Пропускная способность красно-черного дерева на основе транзакционной памяти выше реализации на основе блокировок только при количестве потоков, меньшем или равном 8 и для методов gl_wt и ml_wt выполнения транзакций (рис. 7). Последовательные методы выполнения транзакций (serialirr и serial) уступают блокировкам при любом количестве потоков.

С помощью транзакционной памяти реализованы потокобезопасные красно-черное дерево и хеш-таблица на основе метода разрешения коллизий Hopscotch hashing.

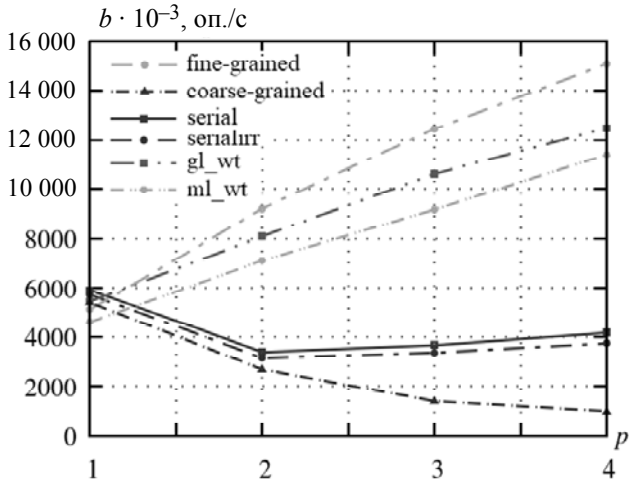

Рис. 4

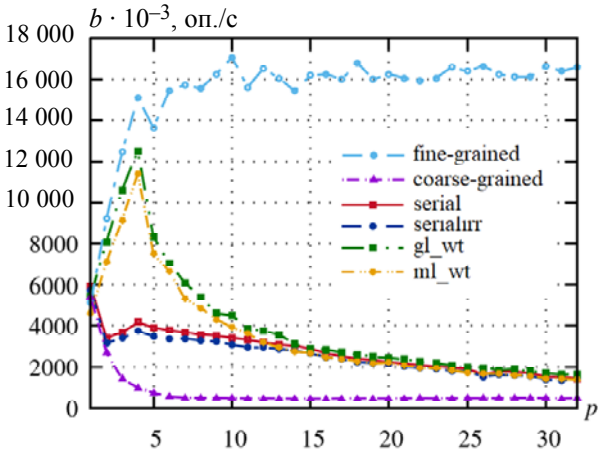

Рис. 5

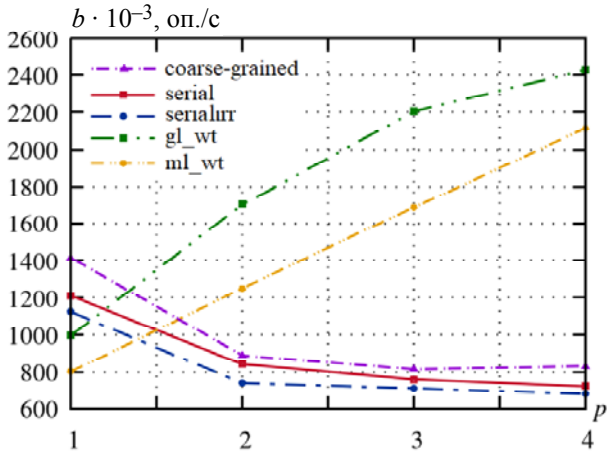

Рис. 6

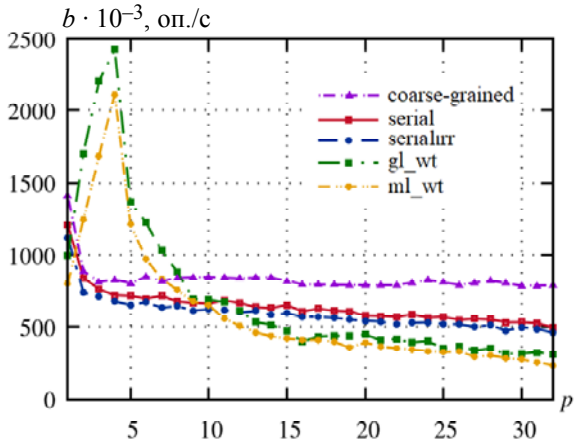

Рис. 7

Моделирование показало, что эффективность хеш-таблицы на основе транзакционной памяти превосходит аналогичные реализации на основе крупнозернистых блокировок. Красно-черное дерево уступает по пропускной способности аналогу на основе мелкозернистых блокировок при реализации с количеством потоков, превышающим количество ядер. Это связано с множеством конфликтов между транзакциями и, как следствие, с их множественными отменами. Рекомендуемым методом выполнения транзакций для хеш-таблицы является метод глобальных блокировок (gl_wt). Большую пропускную способность красно-черного дерева обеспечивают методы выполнения транзакций gl_wt (при количестве потоков, не превышающем числа процессорных ядер) и serial (в случае, если количество потоков превосходит количество процессорных ядер).

СПИСОК ЛИТЕРАТУРЫ

1. Кулагин И. И., Курносов М. Г. Оптимизация обнаружения конфликтов в параллельных программах с транзакционной памятью // Вестн. ЮУрГУ. Сер.: Вычислительная математика и информатика. 2016. Т. 5, № 4. С. 46–60.

2. Kwiatkowski J. Evaluation of Parallel Programs by Measurement of Its Granularity // Parallel Proc. and Applied Mathematics. 2001. P. 145–153.

3. Fraser K. Practical lock freedom / PhD thesis, Cambridge University, 2003. 116 p.

4. Shavit N., Touitou D. Software Transactional Memory // Proc. of the fourteenth annual ACM symp. on Principles of distributed computing, PODC'95. New York, USA, Aug. 1995. P. 204–213.

5. Larus J., Kozyrakis C. Transactional memory // Communications of the ACM. 2008. Vol. 51, № 7. P. 80–88.

6. Conflict Detection and Validation Strategies for Software Transactional Memory / M. Spear, V. Marathe, W. Scherer, M. Scott // Lecture Notes in Computer Science. 2008. Vol. 4167. P. 275–284.

7. Herlihy M., Shavit N., Tzafrir M. Hopscotch Hashing // Proc. of the 22nd Intern. symp. on Distributed Computing, Arcachon, France. Springer-Verlag. P. 350–364.

8. Introduction to Algorithms / T. Cormen, C. Leiserson, R. Rivest, C. Stein. Massachusetts: MIT Press, 2001.

9. Pagh R., Rodler F. F. Cuckoo hashing // J. of Algorithms. 2004. № 51. С. 122–144.

10. Knuth D. Ea. The art of computer programming: fundamental algorithms. Vol. 1 / Addison Wesley Longman Pulishing Co., Inc., Redwood City, CA, USA, 1997.

V. A. Smirnov, A. R. Omelnichenko, A. A. Paznikov
Saint Petersburg Electrotechnical University «LETI»

THREAD-SAFE ASSOCIATIVE ARRAYS, BASED ON TRANSACTIONAL MEMORY

In this paper several algorithms of thread-safe associative arrays (red-black tree, a hash table with open addressing based on the method of Hopscotch hashing collision resolution) using software transactional memory are proposed. Efficiency analysis of associative arrays with different number of used threads and processor cores, comparison with similar data structures based on locks and recommendations for transaction making are described. Basic principles of software transactional memory, different object update politics, conflict detection strategies are introduced. Different lock methods using transactional memory in GCC 5.4.0 are presented. Alternatives used nowadays are reviewed, pros and cons are shown.

Transactional memory, thread-safe data structures, red-black tree, hash table, multi-threaded programming, synchronization multithreaded programs

УДК 303.064

Е. В. Ахунова, Н. Н. Коблов
АО «Научно-производственный центр "Полюс"» (Томск)

Метод автоматизированного формирования и проведения изменений в текстовой конструкторской документации

Предложен метод формирования перечня элементов непосредственно в редакторе PDM-системы, суть которого в том, что источником данных для такого перечня является BOM-файл или файл в формате САПР. При вложении файла в систему происходит запись структуры элементов в виде дерева. Недостающие данные можно добавлять в редакторе PDM-системы. По ним формируется подлинник перечня элементов в формате TIF. При формировании подлинника происходит сравнение с предыдущей версией и в штампах измененных листов делается отметка. Различия по элементам автоматически записываются в извещение об изменении. Таким образом, благодаря автоматическому сравнению версий и проведению изменений значительно сокращается время на разработку конструкторской документации. Кроме того, достоинствами предложенного метода являются повышение точности за счет минимизации расхождения данных между принципиальной электрической схемой и перечнем элементов, отсутствие необходимости в промежуточном программном обеспечении – редакторе. Метод автоматизированного формирования перечня элементов в PDM-системе успешно внедрен и доказал свою эффективность на приборостроительном предприятии ракетно-космической отрасли – НПЦ «Полюс».

Перечень элементов, PDM, САПР, принципиальная электрическая схема

Современные тенденции развития экономики требуют от приборостроительных предприятий ускорения выпуска новых изделий. Обеспечить это может лишь хорошо организованное взаимодействие всех участников производства и максимально высокий уровень автоматизации труда. На предприятии, занимающемся созданием радиоэлектронной аппаратуры, четко просматривается цепочка движения документации: схемотехник – конструктор – технолог – производство. Схемотехник выпускает принципиальную электрическую схему, являющуюся исходной для конструктора, разрабатывающего, в свою очередь, конструкторскую документацию, которую он передает технологу для разработки технологии производства. Таким образом, конструкторская