Software Transactional Memory system for C++

Serge Preis
Intel Corporation
serguei.v.preis@intel.com

Tian Xinmin
Intel Corporation
xinmin.tian@intel.com

Ali-Reza Adl-Tabatabai
Intel Corporation
ali-reza.adl-tabatabai@intel.com

Segey Kozhukhov
Intel Corporation
sergey.s.kozhukhov@intel.com

Ravi Narayanaswamy
Intel Corporation
ravi.narayanaswamy@intel.com

James Cownie
Intel Corporation
james.h.cownie@intel.com

Aleksei Cherkasov
Intel Corporation
aleksei.g.cherkasov@intel.com

Robert Geva
Intel Corporation
robert.geva@intel.com

Abstract

Multicore hardware recently received wide acceptance in all areas of computing from servers to notebooks and now it requires novel software to unleash full power of parallelism. By this reason technologies for parallel programming especially for shared memory systems (such as multicore) are getting significant boost nowadays.

Among hot topics of ongoing researches in this area are easy to use and scalable concurrency control mechanisms and one of those is Transactional Memory (TM). This paper presents pure software implementation of transactional memory system (STM) which supports the following TM ideology of shared data accesses: all data modifications belonging to the same transaction (atomic section) become visible to other threads at the same moment or discarded altogether if conflict with other thread (contention) is detected, in this case transaction is restarted.

The system consists of C/C++ compiler supporting TM-specific language extensions and TM run-time library. This paper primarily focuses on language constructs and aspects of C++ support: the system presented is, perhaps, the fist STM system which consistently supports C++ classes, inheritance, virtual methods, functional and class templates and exception handling

Transactional memory provides simple means for concurrency control and adds such advantages over traditional approaches as composability of transactional codes and failure atomicity. The cost of convenience is performance: TM and especially STM adds overhead on code in transactions. Some optimization helping to relax this problem are listed and discussed. Paper also shows comparative performance results of STM versus different lock techniques.

The main result of the presented work is publicly available full-featured STM system for C/C++ which accelerates development of concurrent systems providing syntax and semantics of single global lock while achieving scalability of fine-grain locking systems and gives some nice additional benefits.

***Keywords**: Software Transactional Memory; Concurrency control; Multithreading; C++ programming*

Система Программной Транзакционной Памяти для C++

Сергей Прейс
Intel Corporation
serguei.v.preis@intel.com

Шинмин Тиан
Intel Corporation
xinmin.tian@intel.com

Али-Реза Адл-Табатабаи
Intel Corporation
ali-reza.adl-tabatabai@intel.com

Сергей Кожухов
Intel Corporation
sergey.s.kozhukhov@intel.com

Рави Нараянасвами
Intel Corporation
ravi.narayanaswamy@intel.com

Джеймс Коуни
Intel Corporation
james.h.cownie@intel.com

Алексей Черкасов
Intel Corporation
aleksei.g.cherkasov@intel.com

Роберт Гива
Intel Corporation
robert.geva@intel.com

Abstract

Многоядерные аппаратные платформы, получившие распространение в последнее время требуют нового программного обеспечения, чтобы раскрыть весь заложенный в них потенциал. В связи с этим технологии параллельного программирования для систем с общей памятью (каковыми являются многоядерные системы) получили последнее время заметное развитие.

В частности, большое внимание уделяется исследованиям в области простых в использовании и эффективных механизмов параллельного доступа к общим данным, одним из которых является Транзакционная Память. В данной статье пойдёт речь о чисто программной реализации системы транзакционной памяти. Транзакционная Память обеспечивает такой доступ к разделяемым данным, при котором изменения данных в рамках транзакции либо становятся видимы другим потокам все сразу, либо (в случае конфликта с другими транзакциями) отменяются и транзакция перезапускается.

Система состоит из компилятора для языков C, C++, расширенных конструкциями ТП, и библиотеки времени исполнения. В статье основной упор сделан на языковые расширения и поддержку C++ – представленная система ТП едва ли не первая для этого языка поддерживающая классы C++, их наследование, виртуальные функции, шаблоны и обработку исключений.

Транзакционная память - удобное средство управления доступом к разделяемым данным, предоставляющее ряд преимуществ: возможность переиспользования транзакционного кода в рамках другого транзакционного кода, атомарность обработки ошибок и т.п. Платой за удобство служит производительность – использование ТП вносит накладные расходы и потому оптимизация производительности – важный аспект представленной работы.

Основным результатом является общедоступная, полнофункциональная система программной транзакционной памяти. Её использование позволит ускорить построение эффективных параллельных систем за счёт сочетания простой программной модели аналогичной неименованным критическим секциям с масштабируемостью, достижимой лишь при использовании индивидуальных блокировок для данных.

***Keywords**: Программная транзакционная память; многопоточное программирование; синхронизация доступа; Программирование на C++*

1. Введение

Распространение многоядерных платформ поставило перед разработчиками ПО новые задачи: чтобы полностью использовать мощность современных вычислительных систем необходимо разрабатывать параллельные алгоритмы, что заметно усложняет и алгоритмическую и архитектурную часть программных систем. В связи с этим средства упрощения разработки параллельных (многопоточных) приложений приобрели особую актуальность, и интерес к ним растёт год от года.

Одной из самых сложных задач в реализации параллельных алгоритмов с общей памятью (таково большинство многопоточных алгоритмов) является задача эффективной реализации доступа к разделяемым ресурсам. С одной стороны необходимо обеспечить корректное функционирование – отсутствие взаимных блокировок (deadlocks) и условий гонок (race conditions), с другой стороны простои из-за ожидания доступа не должны снижать масштабируемость производительности системы с ростом числа потоков/ядер.

Традиционным методом организации доступа к разделяемым данным являются различные виды блокировок (критические секции, семафоры и т.п.). Однако реализация эффективной синхронизации этими средствами может быть достаточно сложна, а отладка и поиск ошибок – ещё сложнее. Наиболее простой стратегией организации синхронизации является использование одной блокировки для всех разделяемых ресурсов, однако в большинстве случаев эта стратегия отрицательно влияет на масштабируемость производительности. Лучшие результаты даёт использование различных блокировок для различных ресурсов, но трудоёмкость этого подхода несравнимо выше.

Транзакционная память (TM – Transactional Memory) – одно из современных перспективных направлений исследований в области организации управления доступом к разделяемой памяти. Суть подхода в том, что все изменения данных изолированы в пределах транзакции и становятся видны либо все сразу на выходе из неё, либо отменяются и вся транзакция перезапускается сначала, если обнаружен конфликт с другой транзакцией.

Синтаксически работа с транзакционной памятью аналогична неименованным критическим секциям, однако, никакой из потоков, одновременно выполняющих транзакцию, не блокируется – все они выполняются параллельно. Двумя дополнительными преимуществами транзакционной памяти являются (1) возможность вложенных транзакций – код с транзакциями может быть переиспользован в рамках другого транзакционного кода; (2) возможность откатить транзакцию явно позволяет вернуть память в корректное состояние при обнаружении ошибки в транзакции.

По причинам, описанным выше, интерес к TM последнее время заметно вырос. Реализация транзакционной памяти требует того или иного мониторинга доступов к памяти, потому большинство работ относятся к управляемым (managed) языкам и системам программирования, таким как Java[9], C#[11], Haskell[10] и Caml[15]. Определённые исследования были посвящены C [17,6] и практически нет работ посвящённых C++ – отдельные аспекты рассматриваются в [1,5], но эти работы не претендуют на полноту. Безусловно, C и C++ остаются одними из самых распространённых языков в областях системного и высокопроизводительного программирования, потому наша работа достаточно актуальна.

Представленная в работе система – это чисто программная (без специальной аппаратной поддержки) реализация транзакционной памяти (STM – Software Transactional Memory) для C и C++. В отличие от предыдущих работ она

1. Поддерживает не только C, но и C++ включая классы, наследование, виртуальные функции, неявные конструкторы и деструкторы, шаблоны, управление памятью и обработку исключений, а также вызовы библиотечных функций, в том числе для ввода-вывода.
2. Основана на существующем высококлассном оптимизирующем компиляторе, предоставляет языковые расширения и набор оптимизаций для транзакционной памяти.
3. Включает высокопроизводительную динамическую библиотеку, поддерживающую несколько различных и переключаемых во время исполнения режимов работы, в том числе сериализованный режим для исполнения библиотечного кода. Интерфейс библиотеки позволяет использовать различные алгоритмы STM и переключать их во время исполнения.
4. Представляет собой законченную и производительную реализацию STM, доступную для тестирования, экспериментов и предварительного использования. Результаты внутреннего тестирования производительности и масштабируемости на наборе тестов SPLASH2 представлены ниже в секции 5.

2. Расширения C++ для ТП

Представленная система расширяет C++ новыми языковыми конструкциями (а не прагмами компилятора, как, например, в [46, 28]) и, в отличие от ранних работ ориентированных только на C [17], поддерживает многие ключевые конструкции и механизмы C++.

2.1. Атомарные блоки

Основной конструкцией для транзакционной памяти, определяющей собственно область транзакционного исполнения, является атомарный блок (atomic block):

Пример 1:

```
__tm_atomic {
 if (local_var > Global_val) {
   Global_max = local_var;
   local_var = Global_arr[++loc_i];
 }
}
```

Аналогично конструкциям **atomic** из ранних работ [1, 18] он определяет, что каждый такой блок исполняется как транзакция: атомарно и изолированно от других таких блоков.

Семантически атомарный блок эквивалентен неименованной критической секции: каждый атомарный блок изолирован и как бы целиком выполняется до или после атомарных блоков в других потоках. В действительности же все они выполняются параллельно, но без эффектов в других потоках. Внутренность атомарного блока трансформируется компилятором таким образом, что все доступы к глобальным данным обрабатываются вызовами STM-библиотеки. Это позволяет обнаруживать *конфликты* – одновременные доступы к одинаковым данным из разных транзакций, причём, как минимум из одной – по записи. В случае конфликта библиотека восстанавливает локально-изменённые данные к состоянию на начало блока и, с новыми значениями глобальных данных, блок перезапускается снова до тех пор, пока не пройдёт без конфликтов (для предотвращения бесконечных откатов применяются специальные меры).

Любая передача исполнения за пределы атомарного блока вызывает попытку успешного завершения транзакции и публикации изменений. Это касается как обычного выхода из блока, так и выполнения конструкций *break, return, goto* и т.п.

Атомарный блок может содержать любые конструкции языка C++, допустимые в блоке кода, включая, другие атомарные блоки.

Пример 2:

```
__tm_atomic {
   // Some code
    __tm_atomic {
      // some more code
    }
}
```

Все изменения сделанные во всём стеке вложенных транзакций станут видимы только при завершении самой внешней транзакции (closed nesting [28]). Однако, откаты, в том числе по запросу от пользователя (см. 2.2.) делается только до границы самой вложенной транзакции, так что границы вложенных транзакций не совсем прозрачны.

Вызовы функций внутри атомарного блока обрабатываются специальным образом: вызовы функции, имеющих транзакционную версию (порождённую компилятором, описанную пользователем или известную библиотечную), заменяются вызовами этой версии. Для косвенных вызовов вставляются динамические проверки наличия транзакционной версии. Прекомпилированные библиотечные функции, функции без транзакционной версии и функции ввода вывода вызывают переход транзакции в сериализованный режим: откат таких функций невозможен и потому для их выполнения должно гарантироваться отсутствие конфликтов, что и достигается сериализацией – глобальной блокировкой всех остальных транзакций.

2.2. Явные (пользовательские) откаты

STM предоставляет расширенный сервис по сравнению с обычными критическими секциями: конструкция **__tm_abort;** откатывает исполнение текущей самой вложенной транзакции, отменяет все сделанные в ней изменения и передаёт управление непосредственно за конец атомарного блока этой транзакции.

Конструкция *__tm_abort* может быть использована лексически только в атомарном блоке, она не может быть использована в функции вызванной из блока. Это ограничение вполне разумно, поскольку код, содержащий атомарный блок с явным откатом, должен быть написан в предположении, что тело атомарного блока может не исполниться.

Кроме того, откат по требованию пользователя невозможен, если до него случился переход транзакции в сериализованный режим. В этом режиме откаты невозможны, потому попытка отката вызовет ошибку времени исполнения, поскольку обнаружить эту проблему при компиляции не всегда возможно.

Однако во вложенной транзакции такой откат допустим.

Пример 3:

```
__tm_atomic {
    cout << "HelloWorld!"; //precompiled
    __tm_atomic {
       // some code
       __tm_abort; // OK, inner TX aborted
     }
     if (some_condition) {
        __tm_abort; // runtime error
   }
}
```

Случаи, когда откат будет гарантированно вызван в транзакции после прекомпилированной функции обнаруживаются компилятором и диагностируются как ошибка, остальные случаи совместного использования в атомарном блоке прекомпилированного кода и явного отката вызывают предупреждение компилятора.

2.3. Разметка функций

Чтобы транзакция не сериализовалась при вызове функции, код функции должен быть обработан компилятором примерно так же, как внутренность атомарного блока. Но поскольку ровно эти же функции могут быть вызваны не из транзакции, то они должны существовать в двух видах – транзакционном (клонированном) и обычном. Для обеспечения раздельной компиляции функций и транзакций, их вызывающих, а также для более точного указания свойств функций, введена возможность разметки функций для транзакционной компиляции.

Идея разметки функций не нова, однако описываемая система предоставляет несколько больше вариантов разметки, чем предыдущие работы [17], давая возможность более гибкого контроля над размером кода и наличием транзакционных версий.

Разметка задаётся с использованием ключевого слова **__declspec()** на Windows™ и **__attribute__(())** на Linux. Примеры даны в варианте для Windows™. x

Пример 4:

```
__declspec(tm_callable) int foo(int);
```

Допускается разметка любых функций, в том числе методов классов и шаблонных функций. Далее приводятся основные пометки для функций. Полный набор пометок описан в [13].

tm_callable – функция, которая может вызываться как из транзакции, так и из обычного кода. Функция полностью дублируется и одна из копий дополнительно обрабатывается компилятором для транзакционного исполнения и переименовывается. Специальные усилия предпринимаются для организации косвенных вызовов транзакционной версии.

Никаких ограничений на содержимое функций, помеченных таким образом, не накладывается, в частности можно вызывать в коде прекомпилированные функции – это вызовет сериализацию текущей транзакции.

Тело такой функции должен компилировать STM-компилятор и её реализация должна быть помечена также как и декларация. В противном случае редактор связей (linker) не найдёт переименованный клон функции, не сможет собрать программу и выдаст ошибку.

tm_pure – функция, которую программист хочет выполнять в транзакции «как есть». Такая функция не должна обращаться к глобальной памяти и иметь побочных эффектов (либо глобальная память константная, а побочные эффекты не важны для исполнения программы). Такая функция, равно как и её вызовы, никак не обрабатываются компилятором, и потому такая пометка может стоять даже на декларации библиотечной функции. Ставя такую пометку, важно понимать, что функция может быть перевызвана многократно в процессе одного исполнения транзакции из-за возможных откатов.

К сожалению, в случае косвенного вызова библиотечная функция, помеченная таким образом, трактуется как прекомпилированная и потому вызывает сериализацию транзакции. Исключение составляют виртуальные tm_pure методы (как прекомпилированные, так и доступные в исходном коде), а также все tm_pure функции с соответствующей пометкой на реализации доступной STM-компилятору.

tm_safe – аналогична *tm_callable*, но предполагает более строгие проверки: в теле функции запрещены вызовы функций, не помеченных явно как tm_pure или tm_safe, а также любые косвенные вызовы кроме виртуальных методов *tm_pure* или *tm_safe*.

tm_safe гарантирует возможность отката и перезапуска, что делает их вызовы безусловно совместимыми с *__tm_abort* в рамках одной транзакции.

2.4. Разметка классов

Разметка класса – это умолчание для разметки всех новых методов класса, включая порождаемые автоматически и виртуальные. То есть разметка класса применяется к методу, если на методе явно не указано другое.

Пример 5:

```
.__declspec(tm_callable) class A {
  __declspec(tm_safe) int getValue(); //safe
  virtual int process(); //tm_callable
};     // A() – implicit, tm_callable
```

Методы, унаследованные из класса предка, сохраняют свою разметку. Невиртуальные методы, переопределяющие методы предка, считаются новыми в классе, и на них действует разметка класса. В тоже время виртуальные методы, переопределяющие методы предка, считаются наследниками (см. 2.5). Это согласуется с правилами наследования/ переопределения методов в C++.

Только *tm_callable, tm_safe* или ***tm_unknown*** может быть указано для класса. Отсутствие пометки, как и на функции, означает отсутствие предположений о природе кода.

Для выборочной отмены пометки на методах *tm_callable/tm_safe*-класса введена ещё одна пометка для методов – **tm_unknown**, кроме того, все перечисленные в 2.2. пометки функций, применимы и к методам классов.

2.5. Наследование разметки

Класс-наследник наследует и пометку класса предка. Для отмены пометки *tm_callable* служит пометка **tm_unknown** указанная на классе. Поскольку пометки на классах влияют только на методы принадлежащее непосредственно этому классу, то допускается произвольная смена разметки в иерархии (в отличие от наследования для виртуальных методов).

В случае множественного наследования для класса выбирается самая сильная разметка из предков. Сила возрастает в ряду *tm_unknown* (нет пометки), *tm_callable, tm_safe*.

Для наследования виртуальных функций действуют более строгие правила. Обработка вызовов виртуальных методов делается по статической информации типа указателя на объект, а он может быть типом предка, поэтому потомок должен иметь не менее сильную пометку, чем предок. В противном случае к вызову могут быть применены более строгие требования, которым потомок не отвечает.

В случае если при переопределении в классе-потомке пометка не указана, она наследуется от метода-предка.

Допустимые сочетания пометок предков и потомков виртуальных функций при переопределении сведены в таблице 1.

В случае множественного наследования из всех возможных пометок для метода выбирается самая сильная (*tm_pure* сильнее, чем *tm_unknown* и противоречит остальным). Если пометки предков противоречивы (напр., *tm_pure* и *tm_callable*), то выдаётся сообщение об ошибке. Выбранная пометка используется как умолчание при переопределении функции в потомке и для проверки совместимости переопределения по таблице 1, приведённой ниже.

Пример 6: (#define – для сокращения строк)

```
#define _ds(x)  __declspec(x)
struct A {
  virtual int _ds(tm_safe) getA() const;
  virtual int process(); //unknown
};
_ds(tm_callable) struct B {
   virtual int getA() const; //tm_callable
   virtual int __ds(tm_unknown) process();
};
struct AB : A, B { //tm_callable
   int _ds(tm_pure) getA(); //tm_safe->error
   int process(); //tm_unknown
};
```

Таблица 1. Правила переопределения

	Derived class			
Base	*unknown*	*callable*	*safe*	*pure*
unknown	yes	Yes	yes	yes
callable	no	Yes	yes	no
safe	no	No	yes	no
pure	no	No	no	yes

2.6. Разметка шаблонов

К шаблонным функциям, методам и классам применимы в полной мере правила разметки для обычных функций, методов и классов соответственно. Кроме того, допускается переопределение разметки базового шаблона на специализации как полной, так и частичной, а также на явной инстанциации шаблонов.

Пример 7:

```
template <class T> _ds(tm_pure) T max(T,T);
template <> char*   //specialization
  _ds(tm_safe) char* max<char*>(char*,char*)
template int*       //instantiation
  _ds(tm_callable) max<int*>(int*,int*);
```

2.7. Обработка исключений

Исключения, не пересекающие границу атомарного блока, обрабатываются по обычным правилам C++ даже если они лежат в атомарном блоке или клонированной версии функции. При этом обработчики, принадлежащие транзакции, обрабатываются компилятором и исполняются транзакционно, как и любой другой код. В частности это означает, что возможна ситуация, когда в обработчике исключения будет обнаружен конфликт и вся транзакция, включая исключение, будет отменена. В этом случае после рестарта исключение может либо возникнуть снова, либо нет. Окончательный результат транзакции будет включать только последнее (бесконфликтное) исполнение, а значит и факт наличия исключения будет определяться только этим последним исполнением.

Если же исключение покидает атомарный блок, то, как и любая передача управления за пределы блока, оно вызывает успешное завершение транзакции и публикацию изменений. Основания для такого поведения следующие:

1. Откат отменит изменения данных в рамках транзакции, и факт наличия исключения будет противоречить данным, анализ причины исключения будет затруднён либо невозможен.
2. Невозможно откатить состояние транзакции, если она к этому моменту исполнила прекомпилированный код и была сериализована. Попытка обработать исключение откатом привела бы к ошибке времени исполнения (см. 2.2).
3. Это полностью согласуется с поведением неименованных критических секций – их состояние не откатывается при исключениях.
4. Существует возможность перехватить исключение и сделать откат транзакции явно с помощью __tm_abort. Явный откат сделает поток управления более очевидным и позволит подготовить слепок состояния транзакции на момент возникновения исключения.

Следует, однако, заметить, что в очередной версии нашей системы будет экспериментальная возможность выполнять откат для исключениях пересекающих границы транзакции.

2.8. Выделение памяти в транзакции

STM-библиотека предоставляет специализированные реализации функций выделения и освобождения памяти для использования внутри транзакции. Это позволяет не сериализовать транзакции с выделением или освобождением памяти внутри и избежать при этом утечек или повторного освобождения памяти. Эта возможность особенно важна для C++, где выделение памяти может быть неразрывно связано с инициализацией объектов. В C предпочтительно выделять память за пределами транзакции, даже если инициализация этой памяти должна делаться внутри. Однако и для C система предоставляет транзакционные версий функций работы с памятью.

3. Оптимизирующий компилятор

Большую часть работы по реализации семантики транзакционной памяти делает STM-библиотека, однако роль компилятора в системе сложно переоценить – он не только предоставляет языковые средства, упрощающие работу с TM, но и играет заметную роль в полной и эффективной её поддержке во время исполнения.

Компилятор в нашей системе реализует самостоятельно или с помощью библиотеки следующие функции:

- Поддержку языковых расширений описанных выше. Именно ясные и простые языковые конструкции играют решающую роль в упрощении реализации доступа к разделяемым данным.
- Поддержку различных режимов работы TM, в том числе требующих дублирования кода функций и отдельных атомарных блоков.
- Расстановку вызовов к STM-библиотеке на границах транзакций и при обращениях к памяти
- Поддержку вызовов функций из транзакции включая:
  - Поддержку прямых вызовов клонированных версий для tm_callable функций;
  - Поддержку косвенных вызовов с динамической проверкой на наличие клонированной версии;
  - Поддержку вызовов tm_pure функций, в том числе и виртуальных методов;
  - Поддержку вызовов прекомпилированных функций с сериализацией транзакций;
- Поддержку обработки исключений;
- Ряд оптимизаций и прочих техник повышения эффективности исполнения, о которых будет сказано ниже.

Чтобы дать библиотеке больше контроля над реализацией STM наша система, в отличие от предыдущих работ [17], использует фиксированный интерфейс с библиотекой без включения элементов реализации непосредственно в порождаемый код. Таким образом, компилятор не может оптимизировать непосредственно код, реализующий алгоритм STM, но за то библиотека может менять стратегии динамически в процессе исполнения или может быть полностью заменена без перекомпиляции приложения. При этом компилятор всё же имеет существенный контроль за тем, как будет исполняться транзакционный код, каковы будут накладные расходы на его исполнение, какие классические оптимизации удастся выполнить до TM-трансформаций. Библиотека предоставляет расширенный набор функций для поддержки оптимизаций.

Чтобы сделать исполнение транзакционного кода наиболее эффективным компилятор реализует следующие стратегии и оптимизации:

1. Вызовы к STM-библиотеке вставляются достаточно поздно в процессе компиляции, чтобы перед этим отработало максимальное число классических оптимизаций.
2. При поддержке библиотеки для повторных обращений к памяти используются специальные оптимизированные функции. Обращения по чтению могут вообще не заменяться вызовами, однако это сужает выбор возможных алгоритмов TM.
3. По транзакции вычисляются и передаются в библиотеку важные свойства, влияющие на выбор алгоритма TM.
4. Обращения к локальной памяти обнаруживаются и не передаются библиотеке, либо передаётся лишь значение для отката.
5. Короткие транзакции могут оставляться «как есть» и исполняться в режиме блокировки для снижения накладных расходов.
6. Известные библиотечные функции, распознанные компилятором, могут получать признак tm_pure.
7. Межпроцедурный анализ позволяет пометить функцию как tm_callable и даже tm_pure по её коду и использовать это при её вызовах.

Разработка оптимизаций ещё не завершена – возможности для дальнейшего повышения производительности далеко не исчерпаны.

4. Немного о TM-библиотеке

Специально для описываемой системы была разработана STM-библиотека, эффективно реализующая самые современные методики TM.

Как было сказано выше, библиотека реализует фиксированный, но достаточно широкий интерфейс, позволяющий реализовать различные алгоритмы и переключать их динамически во время исполнения программы, при этом оставляя компилятору простор для оптимизаций.

Интерфейс содержит примерно следующий набор функций:

```
TxnDesc* getTransaction()
int beginTx(TxnDesc*,int modes_and_hints)
void commitTx(TxnDesc*)
int beginInnerTx(TxnDesc*,int)
void commitInnerTx(TxnDesc*)
void abortTx(TxnDesc*)
void switchToSerialMode(TxnDesc*)
void write<Type>(TxnDesc*,Type*,Type)
Type read<Type>(TxnDesc*,Type*)
void logValue<Type>(TxnDesc*,Type*)
void logMem(TxnDesc*,void*,size t)
void writeAfterRead(Type*,Type)
void writeAfterWrite<Type>(Type*,Type)
Type readAfterRead<Type>(Type*)
Type readAfterWrite<Type>(Type*)
Type readForWrite<Type>(Type*)
```

Назначение большинства функций достаточно понято. Функции вида **After** и **ForWrite* используются для оптимизации повторных обращений к памяти.

На данный момент библиотека реализует 4 основных режима управления TM:

1. С оптимистическими проверками чтений
2. С пессимистическими проверками чтений
3. Приоритетный
4. Сериализованный

Первые два - это классическая транзакционная память с непосредственной блокированной записью и, соответственно, отложенной и непосредственной проверкой чтений. Третий режим позволяет отдать (долгой и конфликтующей) транзакции приоритет во всех конфликтах и таким образом исполнить её до завершения без откатов. Последний режим – глобальная блокировка и серийное исполнение транзакций.

В основном транзакции исполняются в одном из первых двух режимов (в каком конкретно – выбирается динамически), остальные используются для поддержки специальных случаев.

Режим исполнения для транзакции библиотека в основном выбирает сама на основе предыстории исполнения и характеристик, переданных компилятором, однако определённые режимы могут быть затребованы явно для исполнения ввода-вывода или оптимизаций.

Более подробно библиотека описана в [14]

5. Производительность

Тестирование системы производилось на большом числе различных программ, включая SPLASH2 [18] и STAMP [2]. Ниже приводится ускорение исполнения SPLASH2 относительно

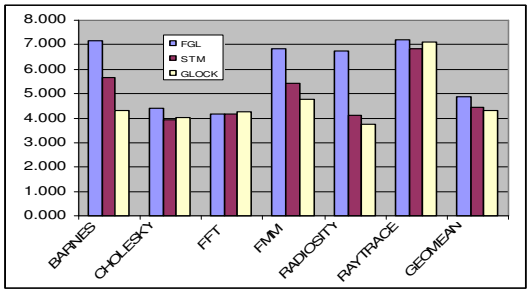

Рис. 1. Ускорение на 8 потоках с оптимизацией

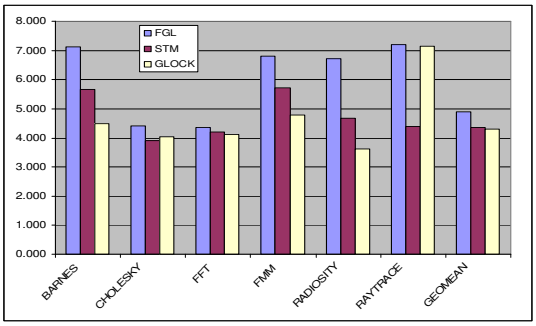

Рис. 2. Ускорение на 8 потоках без оптимизации

однопоточного исполнения для кода с собственными аккуратными блокировками (FGL), одной общей блокировкой (GLOCK) и нашей системой (STM) на 8и-ядерной машине.

Эффект от TM не одинаков для разных приложений. Так в RAYTRACE STM проигрывает даже общей блокировке – все транзакции там короткие и компилятор сериализует их, без этой оптимизации проигрыш заметно больше (см. рис.1). Однако заметно, что накладные расходы на запуск сериализованной транзакции всё же выше, чем при простой блокировке. Этот эффект особенно сильно проявляется когда используются все ядра в системе, так на 16и-ядерной системе для 8и потоков картина будет не столь драматичной. RADIOSITY наоборот страдает от оптимизаций – транзакции в нём разнородные и безо всяких оптимизаций STM показывает лучший результат.

Эффект от использования TM проявляется только при достаточно большом количестве потоков – в нашем случае лишь начиная с 8 потоков на многих приложениях STM начинает ощутимо выигрывать у общей блокировки.

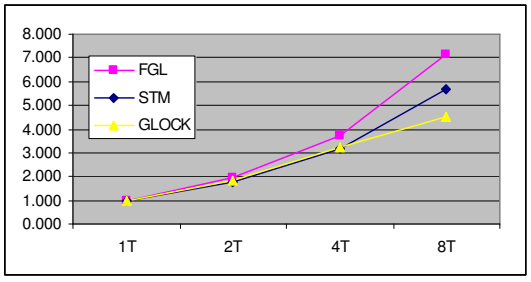

Рис. 3. Ускорение с увеличением числа потоков

6. Заключение

В течение ближайшего месяца общедоступная версия описанного компилятора будет обновлена. Она будет основана на новейшей версии базового компилятора и будет включать исправление ошибок предыдущих версий, поддержку возможностей описанных в статье и даже несколько больше. Но работа продолжается – введение новых оптимизаций позволит ещё поднять производительность кода и устранить многие слабые места, кроме того, ведутся работы по устранению ненужного клонирования и сокращению размера бинарного кода.

С другой стороны всё больше серьёзных игроков на рынке ведут работы в области транзакционной памяти: интерес к TM высказывают Sun Microsystems [5,7], IBM [12], HP [2], AMD[8] и другие. Технология постепенно выходит за рамки чисто научных исследований – пока состояние разработок ещё не позволяет применять TM повсеместно, но постепенно она получает признание: в частности она используется для доступа к общим данным в таких суперкомпьютерных языках как Fortress от Sun[16], X10 от IBM[4] и Chapel от Cray[3].

С ростом числа ядер в массовых системах потребность в TM будет только возрастать, и если удастся устранить основные узкие места, то со временем она вполне может стать одной из основных концепций параллельного программирования.

7. Литература

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard, “Semantics of transactional memory and automatic mutual exclusion”, POPL 2007, pp. 63-74

[2] Boehm, Hans-J; Adve, Sarita V., “Foundations of the C++ Concurrency Memory Model”, PLDI 2008, pp. 68-78

[3] B. Chamberlain, “An Introduction to Chapel: Cray's High-Productivity Language”, AHPCRC/DARPA PGAS Conference 2005

[4] P. Charles, C. Donawa, , K. Ebcioglu, C. Grothoff,, A. Kielstra, C. Von Praun, , V. Saraswat, , and V, Sarkar, “X10: An object-oriented approach to non-uniform cluster computing”. OOPSLA, 2005, pp:519-538

[5] L. Crowl, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, “Integrating transactional memory into C++”. In TRANSACT 2007, Protland, OR

[6] L. Dalessandro, V. J. Marathe, M. F. Spear, M. Scott. “Capabilities and limitations of library-based software transactional memory in C++”. TRANSACT 2007.

[7] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, D. Nussbaum, “Applications of the Adaptive Transactional Memory Test Platform", TRANSACT 2008

[8] AMD, S. Diestelhorst, M. Hohmuth, “Hardware acceleration for lock-free data structures and software-transactional memory”, http://www.amd64.org/fileadmin/user_upload/pub/epham08-asf-eval.pdf

[9] T. Harris and K. Fraser. “Language support for lightweight transactions”. OOPSLA 2003, pp 388-402

[10] T. Harris, S. Marlow, S. P. Jones and M. Herlihy, “Composable memory transactions”, PPoPP 2005,

[11] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. “Optimizing memory transactions”, PLDI 2006. pp 14-25

[12] IBM, “IBM C/C++ for Transactional Memory for AIX, V0.9, Language Extensions and User’s Guide”, hpp://dl.aplhaworks.ibm.com/technologies/xlcstm/xlcstm-whitepaper.pdf

[13] Intel, “Intel® C++ STM Compiler Prototype Edition Language Extensions and Users’ Guide”, http://softwarecommunity.intel.com/isn/Downloads/whatif/stm/Intel-C-STM-Language-Extensions-Users-Guide-V2_0.pdf

[14] Yang Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, , A. Tal, X. Tian, “Design and Implementation of Transactional Constructs for C/C++”, OOPSLA, 2008 (accepted)

[15] M. F. Ringenburg and D. Grossman. “AtomCaml: first-class atomicity via rollback”. ICFP 2005, pp. 92-104

[16] Guy Steele, “The Fortress Parallel Programming Language”, GSPx Multi-core Applications Conf. 2006.

[17] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. “Code generation and optimization for transactional memory constructs in an unmanaged language”, CGO 2007, pp. 34-48

[18] S. C.Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. ISCA 1995: pp 24-36.