УДК 004.432.4 DOI: 10.31114/2078-7707-2020-4-236-243

Средства верификации распределения вычислений в потоковой архитектуре ППВС «Буран»

А.В. Климов

Институт проблем проектирования в микроэлектронике РАН, г. Москва,
klimov@ippm.ru

***Аннотация* — Модель вычислений ППВС «Буран» содержит узлы с индексами, активируемые наборами токенов, приходящих на входы узлов. Токен приходит в ядро (множество ядер), которое определяется при порождении токена по его индексу посредством функции распределения. От правильности работы механизма распределения зависит корректность активаций узлов. В работе формулируются условия корректности функций распределения и описываются статические и динамические средства их обеспечения. Описываемые общие принципы применимы к узлам с любым числом входов, в то время как каждый вход может быть получателем токена, посылаемого на множество ядер.**

***Ключевые слова* — потоковая модель вычислений, потоковый язык, функции распределения, многовходовые узлы, корректность распределения, статическая и динамическая верификация.**

I. Введение

Верификацией мы называем удостоверение в корректности некоторой функциональности (здесь: распределения вычислений по процессорным ядрам) в некотором классе выполнений данной программы (а не на одном или конечном их числе, как при отладочном тестировании). Обычно рассматриваются выполнения с различными (произвольными) исходными данными. В параллельном программировании в процесс могут также вмешиваться различные внешние факторы, такие как: относительные скорости работы подсистем, алгоритмы коммуникационных библиотек, топология сети, количество процессоров, верификация должна показывать корректность при любых таких обстоятельствах.

Средства верификации бывают статические и динамические. Статические работают без выполнения программы с конкретными данными и в конкретных условиях – путем анализа программного кода и доказательства его свойств. Они позволяют делать вывод о правильности программы при любых входных данных (в каком-то классе). Динамические опираются на выполнение кода с конкретными исходными данными с определенной инструментовкой исполнителя, которая включает дополнительные проверки по ходу работы. При их успешном исходе можно утверждать, что результаты для тех же исходных данных будут всегда такими же при любых вариациях конфигурации и параметров определенного класса. В статье рассматриваются вариации распределений, задаваемых посредством функций распределения. Предлагаемые методы существенно опираются на особенности модели вычислений ППВС «Буран». Поэтому мы начнем с подробного и тщательного ее описания.

II. Модель вычислений ППВС «Буран»

Программа представляется как набор описаний узлов, вместе составляющих *граф алгоритма*. Каждое описание узла содержит (в указанном порядке):

- *имя* узла,
- список именованных *входов* (портов) и их типов,
- список имен полей *индекса* и
- *программу* узла.

Обычно индекс — это набор целочисленных величин. Благодаря индексу программа узла может вызываться многократно – при различных значениях индекса. Связи в графе алгоритма образованы операторами посылки токена, которые встречаются в программах узлов и имеют вид:

значение [# *кратность*] → *узел* . *вход* ⟨ *индекс* ⟩ ,

или то же в буквах:

$$D\ [\#m] \rightarrow N.p\ \langle\ I\ \rangle\ .$$

Здесь *значение* D это выражение, вырабатывающее значение, *кратность* m – целочисленное выражение или «#», *узел* N – имя узла, *вход* p – имя входа, *индекс* I – список целочисленных выражений или символов «*» (через запятую). Посылаемый токен имеет логически такую же структуру, только вместо выражений стоят вычисленные значения. Символ «*» в некоторой позиции индекса означает «произвольное значение». Квадратные скобки указывают на возможность отсутствия. Опущенная кратность считается равной 1.

Все выражения в программе узла могут использовать в качестве своих аргументов только локальные переменные, имена входов и имена полей индекса. Есть еще доступ к глобальным константам, которые принимают значение перед началом работы графа и в процессе ее не меняются. Таким образом, множество и содержимое исходящих (из узла с конкретным индексом) токенов может зависеть только от информации, содержащейся во входящих (в данный узел) токенах.

Семантику программы, то есть правила ее выполнения можно задать двумя способами: центральным и распределенным.

A. *Центральный способ*

Абстрактный вычислитель (рис. 1) содержит три части: буфер токенов (БТ), рабочее пространство (РП) и вычислитель узлов (ВУ). Как начальные, так и порождаемые токены поступают сначала в БТ. Оттуда по одному в произвольной очередности они переходят в РП. Несколько токенов, находящихся в РП могут составить активную комбинацию (АК). Соответствующие правила прописываются особо. Типовой вариант – когда группа содержит по одному токену на каждый вход некоторого узла, причем индексы токенов совпадают (во всех позициях). Важно, что свойство группы токенов составлять АК *зависит только от самой группы*, но не от наличия/отсутствия иных токенов в РП.

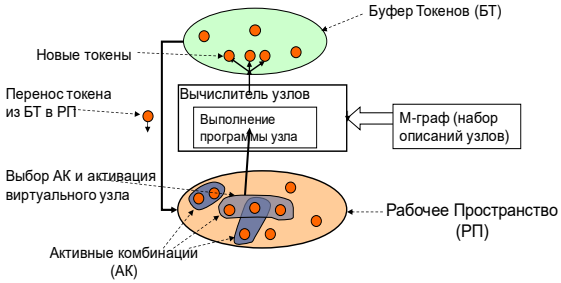

Рис. 1. Абстрактный вычислитель

В некоторый момент в РП может находиться несколько активных комбинаций, возможно пересекающихся. Любая из них выбирается и срабатывает (fire). При этом атомарно (одномоментно) выбранные токены удаляются из РП, и из них формируется пакет, который затем передается в ВУ на выполнение. Пакеты могут передаваться через буфер с какими-то задержками и выполняться в любом порядке или параллельно. Порядок выполнения пакетов в ВУ уже не влияет на эффект, поскольку они не взаимодействуют друг с другом.

Выполнение пакета состоит в выполнении программы узла, при этом входы принимают значения из соответствующих токенов, а индексом становится общий индекс токенов. Результатом является набор новых токенов (быть может, пустой), которые добавляются в БТ. Порядок добавления несуществен: содержимое БТ считается мульти-множеством.

Если в РП нет ни одной АК, то РП считается пассивным. Если ВБ пуст и РП пассивно, то работа завершается, и содержимое РП есть ее результат.

Если токен, участвующий в АК, имел кратность (отличную от 1), то он при срабатывании не удаляется из РП, но из его кратности вычитается 1. Если кратность была бесконечной, она не изменяется. В любом случае при срабатывании АК соответствующая корректировка РП и образование пакета выполняются атомарно.

Бывают и другие причины удержания токена в РП. В рамках статьи нам будет важно только, что согласно правилам токен либо стирается, либо остается (возможно с модификацией). Удержанный токен может участвовать в образовании АК с другими токенами. Но: *каждый конкретный набор токенов может образовать АК не более одного раза*! Это *закон однократной активации*. Как увидим ниже, его обеспечение в реализации трудностей не представляет.

Если в индексе токена содержится символ «*», то он рассматривается как множество возможных индексов, где символ «*» принимает любые значения. При образовании АК индексные множества участвующих токенов должны пересекаться. При этом каждое поле хотя бы в одном токене должно быть определенным, иначе это ошибка. Другими словами, пересечение индексных множеств всех токенов должно быть одноэлементным. Это *закон однозначности* АК. Ниже он сыграет важную роль.

Если в РП было две или более пересекающихся АК, то только одна из них может сработать, после чего оставшиеся снова тестируются на предмет наличия АК. Но если были две или более попарно непересекающихся АК, то результат не зависит от порядка срабатывания и потому они вправе сработать одновременно. Это обстоятельство позволяет организовать распределенное выполнение, разбив РП на независимые части.

Как было сказано выше, токены передаются из БТ в РП по одному. Добавим к этому требование, чтобы перед приемом очередного токена РП было пассивным. Это значит, что после приема очередного токена все новые АК должны быть исполнены (или исчезнуть из-за удалений токенов, участвовавших в других сработавших АК) до приема следующего токена. Это *закон последовательной активации*.

Казалось бы, последний закон уничтожает возможность распараллеливания. Но это не так. Можно организовать разбиение РП на части, которые, работая независимо, обеспечат выполнение, эквивалентное некоторому выполнению с единым РП. Это также вытекает из возможности задать семантику распределенным способом (ниже).

Из перечисленных постулатов следует, что всякая АК непременно содержит последний токен. Это помогает в реализации обеспечить соблюдение закона однократной активации – достаточно обеспечить его в рамках активаций для последнего токена.

Строго говоря, семантика ничего не говорит о параллелизме, но вводит свободу выбора порядка выполнения, считая его последовательным. А параллелизм возникает в реализации как возможный способ выполнения, равносильный последовательному выполнению в некотором допустимом порядке.

Семантика также ничего не говорит о том, как происходит отыскание АК. Считается, что есть некий «оракул», который обязательно ее найдет, если она есть.

Однако, при большом числе входов m у какого-то узла, поиск АК для него среди N токенов может требовать время порядка $O(N^m)$. Точнее, $O(N^{m-1})$, если ищется АК с заданным новым токеном. Эту оценку вряд ли удастся существенно снизить, поскольку в общем случае данная задача NP-полна, если параметр m считать размером задачи. Но для практики это не страшно, поскольку обычно m невелико (не более 3–5), и статически известно. Также обычно статически известны паттерны ключей (в смысле расположения «*») в токенах на каждый вход, на основе которых может быть сгенерирован быстрый специализированный алгоритм поиска АК.

В архитектуре ППВС «Буран» считается, что при двух входах поиск всегда может быть выполнен за один такт работы ассоциативной памяти. При большем числе входов для поиска за один такт потребуется наложить некоторые ограничения (например, отсутствия множественных ключей).

B. Распределенный способ.

Вместо единого РП вводим понятие пространства виртуальных узлов. *Виртуальным узлом* будем называть программный узел с заданными конкретными значениями всех полей индекса. Множество всевозможных виртуальных узлов данной программы образует ее *виртуальное адресное пространство* (ВАП). Совокупность имени узла и значения индекса образует адрес в ВАП, или ключ. Будем обозначать их буквами K, k.

Токены порождаются в одних виртуальных узлах и направляются на другие. Если в индексе во всех позициях стоят конкретные значения, то целевым является конкретный виртуальный узел. Но если хотя бы в одной позиции стоит «*», то данный токен считается направленным на множество виртуальных узлов с произвольным значением на месте «*» в индексе. Будем называть такой токен (как и его ключ) множественным, иначе – одиночным.

Множественный ключ K предствляет множество одиночных ключей (обозначаемое $\widehat{K}$), являющихся его конкретизациями (в этом же смысле одиночный ключ выражает одноэлементное множество). На множестве всевозможных ключей определен частичный порядок $\prec$, соответствующих включению связанных с ними множеств. Ниже мы будем позволять себе отождествлять ключ с соответствующим ему множеством (если это не приводит к недоразумению).

Будем мысленно считать, что в каждом виртуальном узле находится собственный виртуальный процессор для определения условий активации и последующего выполнения программы. При этом множественный токен как бы рассылается на виртуальные процессоры всех мыслимых целевых узлов.

По мере прихода токенов процессор узла накапливает их «на входах» и, когда соберутся по одному (или более) токенов на каждом входе, выполняет активацию. При активации узла с каждого входа снимается по одному токену, формируя из них пакет, который ставится в очередь на исполнение. Токен с кратностью отличной от 1 остается на входе с уменьшенной на 1 кратностью (при бесконечной кратности токен остается без изменения).

Заметим, что корректировка конечной кратности для множественного токена в распределенном способе создает проблему, которой не было в центральном: необходимо вычесть 1 сразу во всех узловых виртуальных процессорах.

Рассмотрим пример. Пусть на двухвходовый узел N посланы токены

a#1→ N.a⟨*⟩
b_0 → N.b⟨0⟩
…
b_9 → N.b⟨9⟩

Когда первым сработает какой-либо из узлов N⟨0⟩,…, N⟨9⟩, кратность токена a станет нулевой, и он должен быть удален, лишая возможности сработать любой из оставшихся. В этом и состоит «зависимость». Зависимости нет, если кратность бесконечная, а также если заранее известно, что количество потенциальных активаций не превосходит кратности. (В последнем случае остается лишь зависимость по стиранию токена при исчерпании кратности числом срабатываний во всех процессорах.)

III. Функции распределения узлов по ядрам

Наличие двух (центральной и распределенной) эквивалентных семантик дает возможность их как бы «скрестить» в рамках физически распределенной реализации на нескольких процессорных ядрах. При этом каждое ядро является полнофункциональным процессором, реализующим центральную семантику. Но на него возлагается ответственность лишь за некоторую часть виртуального адресного пространства, причем части разных ядер не пересекаются. Этого можно добиться, задав функцию распределения (ФР), которая зависит от адреса (ключа) и вырабатывает номер ядра: $p = F(K)$.

Теперь по выходу из ВУ новый токен проходит через блок вычисления ФР, где определяется номер ядра, и направляется через коммутатор токенов в БТ ядра с вычисленным номером. Поскольку все токены, посылаемые на разные входы одного узла имеют один и тот же ключ, то значение ФР для них будет одинаковым, и поэтому они попадут в одно и то же ядро, где и произойдет активация. Но если среди них есть множественные токены, то все не так просто, и об этом пойдет речь ниже.

Функцию распределения [2] выбирает программист, стремясь улучшить (ускорить) прохождение его задачи. При этом приходится искать компромисс между равномерностью (балансом) распределения вычислительной нагрузки на ядра с одной стороны и минимизацией коммуникационных потоков между ядрами с другой. Еще один важный фактор, который также стоит минимизировать, это коммуникационная высота задачи: она определяется как наибольшая длина

пути от входных данных до результата в графе задачи, в котором учитываются только передачи между ядрами, а передачи внутри ядер не в счет. Он, хотя и связан со вторым, но не сводится к нему.

ФР должна быть простой, поскольку она будет вычисляться для каждого нового токена. Ее можно задать формулой, состоящей из операций арифметики с фиксированной запятой ограниченной разрядности, логическими поразрядными операциями (над двоичными представлениями), различными перестановками битов и некоторых дополнительных библиотечных операций.

Обычно фактическое число ядер определяется в последний момент перед началом счета. Поэтому желательно уметь задать некую *базовую* функцию распределения $\varphi(K)$, из которой будет легко получать конкретный вариант $F(K)$ под любое нужное число ядер. В ней уже должны быть учтены такие свойства как равномерность и локальность, которые должны сохраняться при переходе к конкретному числу ядер.

С учетом названных требований целесообразно считать значение $\varphi(K)$ не целым числом, а дробным в интервале $[0; 1)$. Он имеет представление в виде строки битов с точкой слева (а не справа как у целых). Значение $\varphi(K)$ будем называть номером *виртуального ядра*. К номеру реального ядра можно перейти, применив к номеру виртуального ядра некоторую проекцию π (с параметром NP), например, умножение на число ядер и взятием целой части:

$$F(K) = \pi_{NP}(\varphi(K)) = \lfloor NP * \varphi(K) \rfloor. \qquad (1)$$

Когда NP есть степень 2, номер ядра образуется из соответствующего количества старших битов $\varphi(K)$.

Будем считать, что $\varphi(K)$ имеет фиксированную разрядность q. Строка $x_1 \dots x_q$, где $x_i \in \{0,1\}$, представляет числовое значение $x = \sum_1^q x_i 2^{-i}$. Среди 2^q номеров не обязательно должны использоваться все. Но желательно, чтобы так было для некоторого начала $x_1 \dots x_r$, $r \le q$ и при том достаточно равномерно. Это обеспечит равномерность нагрузки при любых $NP \le 2^r$.

Пример. Пусть индекс состоит из двух полей: (t, j), где $j \in [0; Nj]$ – номер ячейки, $t \in [0; Nt]$ – номер итерации. Допустим, в задаче есть узел $X\langle t, j\rangle$, который получает токены от узлов $X\langle t, j-1\rangle$, $X\langle t-1, j\rangle$ и $X\langle t-1, j+1\rangle$. Хорошей базовой функцией распределения будет

$$\varphi(t, i) = \{(t + i)/D\}, \qquad (2)$$

где D – некоторая константа (желательно степень 2), выбираемая с учетом размеров задачи (Nt, Nj). Заключительное значение должно быть чистой мантиссой (с нулевой целой частью), поэтому внешние фигурные скобки, обозначающие взятие дробной части, могут опускаться. Теперь количество реальных ядер NP может независимо выбираться в широких пределах от 1 до D. В каждое ядро попадет косая полоса шириной D/NP.

Среди специальных функций типа «перестановок битов» отметим функцию *zip*. Она берет два (или более) аргументов, и строит результат, беря циклически по одному биту из каждого аргумента. Она сворачивает двухмерный (или более) индекс в одномерный, соблюдая при этом локальность: близкие в многомерном смысле значения переходят в близкие же на одномерной шкале.

Сохранение локальности важно, когда токены передаются преимущественно между узлами с близкими индексами. Оно позволяет эффективно использовать иерархическое строение коммутатора, которое обеспечивает более быструю (и с большей пропускной способностью) передачу токенов между элементами одного компонента, нежели между разными компонентами, причем на всех уровнях. При этом одна и та же функция φ, соблюдающая локальность, будет показывать хорошую эффективность при любой структуре сети, коль скоро эта структура иерархична. Этот эффект подобен написанию так называемых cache-oblivious алгоритмов [3]. Здесь он достигается правильным выбором функции распределения, сохраняющей локальность. Функции *zip* и *unzip* в этом деле очень полезны.

В процессорных ядрах ФР могут вычисляться различными способами:

а) программно: непосредственно в ВУ как часть программы узла, вставляемая компилятором;

б) аппаратно: отдельным специализированным устройством со своей «системой команд»;

в) FPGA-блоком с прошивкой, порождаемой специальным компилятором.

Хорошо, если выбор ФР не влияет на работу программы. Тогда для ее отладки достаточно добиться ее правильной работы на одном ядре, а на любом другом числе ядер и при любой ФР все будет выполняться автоматически. Ниже будут сформулированы требования к ФР, при которых эта цель достигается.

Согласно закону однозначности, токены одной АК всегда имеют единственный общий ключ (даже если все участвующие в ней токены множественные). Если все токены одиночные, то результат не может зависеть от выбора ФР, поскольку токены одной АК любая заданная ФР заведомо отобразит в одно и то же ядро, где они и встретятся. Но если есть множественные токены, то могут возникнуть два вопроса:

i. Будет ли такой токен доставлен во все ядра, в которые попадают АК с его участием.
ii. Не попадут ли токены АК в лишнее ядро, где возникнет повторное срабатывание узла.

Для множественных токенов требуется уточнить понятие функции распределения.

IV. Распределение множественных токенов

Обозначим множество всех множественных ключей через $\mathcal{K}$. Сам ключ будем обозначать прописной латинской K (с возможным индексом), а когда

известно, что ключ точный, то строчной латинской k. Для токена с множественным ключом $K \in \mathcal{K}$ естественно ожидать, чтобы значением $F(K)$ было не одно ядро, а множество (номеров) ядер. В нем должно содержаться всякое ядро, в которое попал бы токен с конкретным ключом k, являющимся конкретизацией ключа K:

$$\forall k \in K: F(k) \in F(K). \quad (3)$$

Если все конкретизации множественного ключа K данная ФР F переводит в одно и тоже ядро:

$$\forall k_1, k_2 \in K: F(k_1) = F(k_2),$$

то само это общее значение и будет значением для K. А в общем случае значение ФР должно быть множеством номеров, которое надо как-то представить. При этом условие (3) позволяет брать «с избытком».

При передаче токена через коммутатор номер ядра используется маршрутизаторами для навигации. Если задано множество мест назначения, то и навигация должна обеспечить рассылку на все ядра из заданного множества (мультикастинг). Поскольку работа рутеров не должна зависеть от программы (графа алгоритма), то и структура множеств должна быть универсальной, не зависящей от программ. С другой стороны, она должна быть достаточной для нужд разных программ. Будем задавать множества, опираясь на двоичное представление номеров, при этом любой разряд может быть задан как «любой» в виде «*». Будем называть такие номера множественными.

Если K – множественный ключ, то функция $\varphi(K)$ вычисляет множество виртуальных номеров в виде строки $s = s_1 \dots s_q$, где $s_i \in \{0,1,*\}$. Выражаемое ею множество конкретных номеров (строк) обозначим $\hat{s}$. Множествами такого вида будем аппроксимировать сверху любые множества конкретных (точных) номеров (двоичных строк).

Пример. Пусть для узла $N\langle i,j\rangle$ задана функция распределения $\varphi_N(K) = zip(i,j)/D$, где $D = 2^{16}$. Пусть $q = 10$. Тогда токен $d \longrightarrow N.p\langle 100,* \rangle$ будет рассылаться на множество виртуальных ядер, изображаемых строкой '*0*1*1*0*0'.

Для двух множественных строк определим отношение $\prec$: $a \prec b$ означает, что $\hat{a} \subseteq \hat{b}$. При этом

$$a_1 \dots a_q \prec b_1 \dots b_q,$$

если для всех $i = 1..q$ имеет место $a_i = b_i$ или $b_i = *$. Отношение $\prec$ является частичным порядком.

Для множественных строк естественным образом определим операции пересечения $\wedge$ и объединения $\vee$, при этом будет выполняться

$$\widehat{x \wedge y} = \hat{x} \cap \hat{y}, \qquad \widehat{x \vee y} \supseteq \hat{x} \cup \hat{y}.$$

То есть пересечение множеств является точным, а объединение – приближенным сверху. Пустое пересечение обозначим как $\perp$ («дно»). Множество всех множественных строк длины q образует решетку [4].

Чтобы было удобнее задавать функции $\varphi(K)$, определим все используемые при этом операции так, чтобы они принимали аргументы-множества, и вырабатывали результаты-множества. Для всякой используемой операции o должна выполняться следующая импликация (*монотонность*):

для любых $a_1, b_1, a_2, b_2, \dots$:

$$a_1 \prec b_1, a_2 \prec b_2, \dots \Rightarrow o(a_1, a_2, \dots) \prec o(b_1, b_2, \dots),$$

где a_i, b_i – множественные ключи. Тогда очевидно, что и для функции φ, являющейся их композицией, будет так же. Большинство арифметических операций будут обычно выдавать «все *», если хотя бы в одном аргументе есть хоть одна «*». С другой стороны, логические поразрядные и переставляющие биты операции могут работать с точностью до бита.

При переходе от базовой функции $\varphi(K)$ к конкретной $F(K)$ важно не слишком расширять множества, чтобы избежать лишних передач. Формула (1) в этом смысле плохая, если NP не степень 2.

V. Условия корректности

Вопросы i-ii теперь мы можем сформулировать математически. Пусть активная комбинация образована m токенами с ключами K_i. И пусть $k \in \bigcap_{i=1}^{m} K_i$ (по закону однозначности такое k единственное).

i. Верно ли для всех i от 1 до m, что $F(k) \in F(K_i)$?

ii. Верно ли, что $\left|\bigcap_{i=1}^{m} F(K_i)\right| = 1$?

Очевидно, что если функция F монотонная, то есть

$$\forall x, y \in \mathcal{K}: x \prec y \Rightarrow F(x) \prec F(y), \quad (4)$$

то на вопрос i ответ всегда утвердительный. Действительно, если АК имеет общий ключ k, то ключ каждого ее токена K_i содержит k, то есть $k \prec K_i$, отсюда по монотонности $F(k) \prec F(K_i)$, то есть $F(k) \in F(K_i)$.

Для положительного ответа на вопрос ii этого недостаточно: образ множества может содержать избыточные элементы. В результате токены одной АК могут встретиться повторно в другом, ложном ядре и породить повторную активацию узла.

Рассмотрим для примера узел N с тремя входами: a, b, c. Пусть три токена, направленные на входы N.a$\langle K_a \rangle$, N.b$\langle K_b \rangle$, N.c$\langle K_c \rangle$ образовали активную комбинацию. Как требует закон однозначности, пересечение $K_a \cap K_b \cap K_c$ является точным ключом, или одноэлементным множеством. Тогда, если задана монотонная функция распределения φ, то пересечение

$$\varphi(K_a) \cap \varphi(K_b) \cap \varphi(K_c)$$

будет по меньшей мере непустым. Но будет ли оно одноэлементным, сказать нельзя. Однако, оно должно быть одноэлементным. Иначе при некотором числе ядер найдутся два разных ядра, в каждое из которых попали все три токена. В результате узел сработает дважды, что приведет к некорректной работе.

Зададимся вопросом: каким свойством должна обладать функция F, чтобы всегда соблюдалось условие ii. Хорошо, если это свойство удается обеспечить построением φ, или доказать из построения. Будет, например, достаточно, чтобы функция F сохраняла пересечение, точнее пересечение $\cap$ ключей переводила бы в пересечение $\bigwedge$ множественных строк. Заметим, что в выбранном представлении множеств ключей или номеров ядер оно всегда выполняется точно: пересечение представимых множеств всегда представимо. Для представления пустого пересечения введем элемент $\perp$ («дно»). Для любого $k \in K$ пусть $\perp \prec k$. Монотонность F требует, чтобы $F(\perp) = \perp$. Однако, теперь оговорим, что сохраняться должно лишь непустое пересечение:

$$\forall x, y \in \mathcal{K}: x \cap y \neq \perp \Rightarrow F(x \cap y) = F(x) \bigwedge F(y). \quad (5)$$

Без этой оговорки требование (сохранения пересечения) слишком жесткое: из него следовало бы, что разные узлы должны быть в разных ядрах. С ней же в него войдут по крайней мере функции типа «перестановки битов», у которых каждый бит результата зависит не более чем от одного бита аргумента. Более того, учитывая, что не любые множественные ключи возможны в данной программе, будет достаточно, чтобы каждый бит результата зависел лишь от одного фрагмента ключа, где фрагментом считается такая часть ключа (поле или группа полей), которая в каждом токене (ключе) либо полностью задана, либо полностью выражена как «*». В предположении, что любые ключи с таким условием возможны, это условие является также и необходимым, что показывает следующая Лемма.

Лемма. Пусть функция F монотонна, сохраняет $\perp$ *и переводит точные значения в точные. Тогда следующие условия эквивалентны:*

- *a) Каждый бит результата зависит не более чем от одного фрагмента аргумента.*
- *b) Функция F сохраняет точное пересечение любых двух (множественных) ключей.*
- *c) Функция F сохраняет точное пересечение любого числа (множественных) ключей.*

Доказательство:

c)⇒b). Тривиально: b) есть частный случай c).

b)⇒a). Мы называем пересечение *точным*, когда оно состоит из единственного точного значения. Предположим противное: пусть есть бит результата, зависящий по крайней мере от двух фрагментов, назовем их A и B, а все остальные – C. Рассмотрим два множественных ключа (с точностью до взаимного расположения фрагментов): ⟨a,*,c⟩ и ⟨*,b,*⟩, где a,b,c – такие конкретные значения соответствующих (одноименных) фрагментов, при которых реализуется зависимость от фрагмента B для первого ключа, и зависимость от фрагмента A для второго ключа (во втором случае в части C должна стоять некоторая константа c_1, но мы заменяем ее на «*», и по монотонности F это сохранит «*» в данном бите результата). Очевидно, их пересечение точное – ⟨a,b,c⟩, но результат F от каждого имеет «*» в данном бите (по предположению). Но это противоречит посылке *b).*

a)⇒c). Предположим противное: имеется набор из $m \geq 2$ ключей $K_1, \dots, K_m$, которые пересекаются на точном ключе k, но их образы $F(K_i)$ пересекаются на множественном значении, имеющем «*" в каком-то разряде (случай $m = 1$ тривиален). Согласно посылке *a)* этот бит зависит лишь от одного фрагмента. Исходя из принципа, что отображение F строит наиболее тесное значение результата, покрывающее все точные образы точных конкретизаций аргумента, мы приходим к выводу, что каждый K_i должен иметь «*» в найденном фрагменте (иначе для этого K_i выделенный бит был бы конкретным: 0 или 1), что противоречит точности их пересечения.

В действительности можно также доказать, что функция F, удовлетворяющая условиям *a)–c)* Леммы будет сохранять вообще любое непустое пересечение. Но для наших практических целей это уже не нужно: нам достаточно, что она сохраняет точное пересечение любого числа аргументов, что и обеспечивает соблюдение условия ii.

Условие a) Леммы дает удобный критерий полной корректности (когда выполняются оба условия i и ii), а именно: *каждый бит значения зависит не более чем от одного фрагмента ключа*. Он легко может быть проверен статически по формулам, определяющих функцию. Такие функции устроены следующим образом: сначала от каждого фрагмента формируется, как функция одного аргумента, некоторое значение в виде двоичного кода, а затем из полученных кодов составляется итоговое значения путем перестановки их битов. Статическую проверку следует проводить для каждого программного узла отдельно.

Основанный на Лемме метод проверки корректности ФР предполагает, что при работе программы могут встретиться любые множественные ключи, соблюдающие фрагменты. Однако при учете реально используемых в тексте потоковой программы ключей распознавание корректных ФР может быть улучшено. Более точный и полный алгоритм анализа с учетом используемых в программе шаблонов ключей предстоит разработать.

Заметим, что если хотя бы в одном токене ключ K_i точный, то для него значение $F(K_i)$ тоже будет точным, а значит, условие ii для такой АК соблюдается и без опоры на свойство (5). Это свойство актуально только для случая, когда ключи всех токенов одной АК множественные. Эту ситуацию можно считать m-кратным групповым узлом – обобщением двойного группового узла. Для таких узлов раньше считалось, что только один из двух токенов может быть распределенным, то есть глобальным. Мы расширяем возможности распределения для таких узлов, позволяя распределять оба токена. Также теперь эта ситуация обобщается на случай любого числа входов. В прежнем расширении ППВС многовходовыми узлами [5] все их токены должны были иметь точный ключ.

VI. Динамический контроль

Описанные выше требования и условия дают «пищу» для средств динамического контроля. Рассмотрим различные возможности для выполнения динамических проверочных тестов.

A. *Тест однозначности ключа при активации.*

Перед активацией узла сразу после успешного сопоставления проверяем условие

$$\left|\bigcap_{i=1}^{m} K_i\right| \leq 1. \quad (6)$$

Здесь $K_i, i = 1..m$ – набор ключей токенов, образующих активную комбинацию m-входового узла. Фактически проверяется равенство, поскольку успех сопоставления означает, что пересечение не пусто, и значит число элементов в нем положительно. Это требование закона однозначности АК. Для выполнения этой проверки достаточно выполнить логическое И для масок всех ключей K_i (маска ключа имеет 1 в бите, лежащем в элементе со значением * и 0, когда значение элемента задано). При нарушении – авост.

B. *Тест монотонности ФР – условие i*

При успешном сопоставлении проверка монотонности ФР F могла бы состоять в сравнении значения ФР для ключа-пересечения $k \in \bigcap_{i=1}^{m} K_i$ со значением ФР каждого не точного ключа K_i:

$$F(k) \prec F(K_i). \quad (7)$$

Для этого придется в токенах сохранять значение $F(K_i)$, которое используется для навигации и дополнительно вычислять $F(k)$. Однако, сделать это не просто, поскольку блок вычисления ФР находится при выходе из ИУ, а вычислять требуется внутри СП.

C. *Тест однозначности номера ядра – условие ii*

Этот тест имеет смысл только для АК, в которой все ключи – множественные (m-кратный групповой узел). Проверяем условие

$$\left|\bigcap_{i=1}^{m} F(K_i)\right| \leq 1. \quad (8)$$

Проверка сводится к логическому И масок значений ФР всех токенов АК. При нарушении – авост. Здесь также фактически проверяется равенство, поскольку в пересечении содержится номер текущего ядра.

В классической архитектуре ППВС [2] функция распределения F может выдавать либо точный номер ядра, либо * («все ядра»). Последний случай называется «глобальный токен» – он рассылается на все ядра. Когда $m \leq 2$ (все узлы не более чем двухвходовые) тест С вырождается в проверку того, что два сопоставленных токена не являются одновременно глобальными. Иными словами, тест С есть обобщение этой проверки на случай любого числа входов и ФР, дающих множества номеров ядер более общего вида.

D. *Тест однозначности номера ядра с коррекцией*

Однако в тесте С мы можем не объявлять авост, сделав дополнительную проверку. Пусть условие (8) ложно, тогда перед активацией данной АК с ключом $k = \bigcap_{i=1}^{m} K_i$ вычислим $F(k)$ и сравним результат с текущим номером ядра $F(k) = p_{current}$. Если отличается, активацию просто отменим. Тем самым активация будет выполнена только один раз в ядре, номер которого есть $F(k)$. Здесь, однако, имеется та же трудность, что и тесте В.

E. *Тест однозначности номера виртуального ядра*

Теперь рассмотрим представление функции F в виде композиции: $F(K) = \pi_{NP}(\varphi(K))$, где φ выдает номер виртуального ядра и задается программистом под задачу, а π может зависеть от числа реальных ядер и определяется системой (конфигурацией). Можно один раз задать функцию $\varphi(K)$, а затем менять π_{NP} в зависимости от конфигурации мультипроцессора и числа ядер *NP*. Обе функции задаются как композиции элементарных функций, сохраняющие точные значения и отношение порядка $\prec$. Поэтому и сами они будут такими же по построению. От функции φ потребуем однозначность, а π может ее нарушать и это может корректироваться. Величина φ вычисляется на выходе из ИУ, вычисление π может быть распределенно встроено в коммутатор токенов.

В структуре токена надо предусмотреть место для $\varphi(K)$ размером $2q$ бит (достаточно q троичных разрядов). Перед активацией АК вычислим

$$v = \bigcap_{i=1}^{m} v_i \text{, где } v_i = \varphi(K_i). \quad (9)$$

Проверим, что $|v| = 1$. (Иначе авост: φ не сохраняет точное пересечение.) При этом множество $\bigcap_{i=1}^{m} \pi_{NP}(\varphi(K_i))$ может не быть точным, поскольку π_{NP} не обязано сохранять точность пересечения, и это может привести к повторной активации. Поэтому теперь вычислим $p = \pi_{NP}(v)$ и проверим $p = p_{current}$. Если равны, то ОК, иначе активацию отменяем. Этот вариант требует от блока сопоставлений умения вычислять только π_{NP}, что проще, поскольку она определяется аппаратной конфигурацией, в отличие от задаваемой программистом φ. Применение данного теста мотивируется следующей теоремой.

Теорема 1. *Пусть при выполнении программы P с базовой функцией распределения φ на некотором числе ядер NP все активации прошли тест* E. *Тогда на любом другом числе ядер и тех же исходных данных выполнение будет завершено с тем же результатом. Когда известно, что φ монотонна, проверку $|v| = 1$ можно свести к проверке $|v| \leq 1$.*

VII. Заключение

Была рассмотрена модель вычислений, являющаяся обобщением модели вычислений, реализуемой ППВС «Буран». Обобщение осуществлено в направлениях:

1. Многовходовые (более 2-х входов) узлы с любым числом входов, допускающие множественные токены на каждом входе. При этом пересечение ключей должно быть точным.
2. Функция распределения $\varphi: \mathcal{K} \longrightarrow \mathcal{P}$ монотонно отображает множественные ключи в множества процессоров. Определено условие корректности ФР: сохранение точного пересечения.
3. Описан механизм динамического контроля, для проверки корректности ФР в динамике.
4. Предложен механизм исправления некорректностей ФР в динамике, подавляющий дублирующие активации узлов при некорректных ФР.

Вопросы эффективной реализации предложенных обобщений будут рассмотрены в будущих статьях.

ЛИТЕРАТУРА

[1] Стемпковский А.Л., Левченко Н.Н., Окунев С.А., Цветков В.В. Параллельная потоковая вычислительная система — дальнейшее развитие архитектуры и структурной организации вычислительной системы с автоматическим распределением ресурсов // Информационные технологии. 2008. №10. С. 2-7.

[2] Змеев Д.Н., Климов А.В., Левченко Н.Н. Средства распределения вычислений в ППВС «Буран» и варианты реализации блока выработки хэш-функций // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). 2016. № 2. С. 107-113. URL: http://www.mes-conference.ru/data/year2016/pdf/D115.pdf

[3] Frigo, M., Leiserson, C. E., Prokop, H., Ramachandran, S. Cache-oblivious algorithms // In: 40th Ann. Symp. on Foundations of Computer Science (FOCS'99). IEEE Computer Society, Washington, DC. 1999. P. 285-297.

[4] Биркгоф Г. Теория решеток. М.: Наука,1984.

[5] Левченко Н.Н., Окунев А.С., Яхонтов Д.Е., Шурчков И.О. Варианты реализации контроллера памяти параллельной потоковой вычислительной системы для работы с векторными и многовходовыми узлами // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). 2012. С. 463-466. URL: http://www.mes-conference.ru/data/year2012/pdf/D100.pdf

Tools for Verification of Computation Distribution in the Parallel Dataflow Computing System (PDCS) “Buran”

A.V. Klimov

Institute for Design Problems in Microelectronics of RAS, Moscow, klimov@ippm.ru

***Abstract* — The computation model of parallel dataflow computation system (PDCS) “Buran” is comprised of nodes with indexes (tags) activated by sets of tokens arriving at the node’s input ports. A token arrives to the core (the set of cores) which is determined from token index by the distribution function when token is created. The correctness of the node activation depends on the correct operation of the distribution mechanism. The paper formulates the correctness conditions for the distribution functions and describes the static and dynamic means of ensuring them. The described general principles are applicable to nodes with any number of ports, while every port can be the receiver of a multicast token.**

***Keywords* — dataflow computation model, dataflow language, distribution functions, multi-port nodes, distribution correctness, static and dynamic verification.**

REFERENCES

[1] Stempkovskij A.L., Levchenko N.N., Okunev A.S., Cvetkov V.V. Parallelnaya potokovaya vychislitelnaya sistema — dalnejshee razvitie arkhitektury i strukturnoj organizacii vychislitelnoj sistemy s avtomaticheskim raspredeleniem resursov (Parallel dataflow computing system — further development of the architecture and structural organization of the computing system with automatic allocation of resorces). Zhurnal «INFORMACIONNYE TEHNOLOGII». 2008. № 10. P. 2-7 (in Russian).

[2] D.N. Zmejev, A.V. Klimov, N.N. Levchenko. The Tools for Computation Distribution in the PDCS "Buran" and Hash-Functions Block Implementation Options // «Problemy razrabotki perspektivnyh mikro- i nanojelektronnyh sistem» (MES). 2016. № 2. P. 107-113 (in Russian).

[3] Frigo, M., Leiserson, C. E., Prokop, H., Ramachandran, S. Cache-oblivious algorithms // In: 40th Ann. Symp. on Foundations of Computer Science (FOCS'99). IEEE Computer Society, Washington, DC. P. 285-297. 1999.

[4] Birkhoff, G. Lattice Theory. Providence, Rhode Island, 1967.

[5] Levchenko N.N., Okunev A.S., Yakhontov D.E., Shurchkov I.O. Implementation Options for the Memory Controller of a Parallel Dataflow Computing System for Operation with Vector and Multi-port Nodes. «Problemy razrabotki perspektivnyh mikro- i nanojelektronnyh sistem» (MES). 2012. P. 463-466 (in Russian).