УДК 681.3.06

РАСПРЕДЕЛЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ РЕСУРСОВ В СРЕДАХ С МНОГОПОТОЧНОЙ РЕАЛИЗАЦИЕЙ ГИПЕР-АВТОМАТА*

В.Е. Зюбин
Институт автоматики и электрометрии СО РАН
Россия, 630090, Новосибирск, просп. акад. Коптюга, 1
E-mail: zyubin@iae.nsk.su

А.Д. Петухов
Институт автоматики и электрометрии СО РАН
Россия, 630090, Новосибирск, просп. акад. Коптюга, 1
E-mail: petuhov@iae.nsk.su

Ключевые слова: гипер-автомат, язык РЕФЛЕКС, логический параллелизм, управляющие алгоритмы, событийные архитектуры, время реакции системы на внешнее событие, балансировка нагрузки
Key words: hyper-automaton, REFLEX language, logical parallelism, control algorithms, event-driven architectures, external event response time, load balancing

Рассмотрена проблема балансировки вычислительной нагрузки в средах многопоточной реализацией гипер-автомата. Обсуждаются методы априорной оценки времени реакции системы на внешнее событие. Предложено расширение синтаксиса языка РЕФЛЕКС и показано, что оно не противоречит семантике языка. Перечислены функциональные требования к программному обеспечению.
LOAD BALANCING IN MEDIA WITH MULTITHREAD IMPLEMENTATION OF HYPER-AUTOMATON / V.E. Zyubin (Institute of Automation and Electrometry SB RAS, prosp. acad. Koptyuga, Novosibirsk 630090, Russia, E-mail: zyubin@iae.nsk.su), A.D. Petukhov (Institute of Automation and Electrometry SB RAS, prosp. acad. Koptyuga, Novosibirsk 630090, Russia, E-mail: petuhov@iae.nsk.su). Problem of load balancing in media with multithread implementation of hyper-automaton is examined. A-priory calculation techniques of external event response time is discussed. Semantically consistent extension for the REFLEX language is suggested. Functional specification for software is made.

* – в оригинальную версию статьи внесены незначительные правки -
Зюбин В.Е. 31 августа 2004 г.

1. Введение

На начальных этапах проектирования цифровых систем управления разработчику необходимо выбрать целевую вычислительную платформу и конфигурацию технических средств, соответствующие заданным требованиям. Одно из ключевых требований - обеспечить приемлемое время реакции системы на внешнее событие. При отсутствии этого условия *спроектированное* теряет право называться системой управления. Общепринятый подход - выбирать платформу «на глазок», что часто приводит к дополнительным финансовым

затратам. Отказаться от этой порочной практики мешает сложность оценки ресурсоемкости алгоритма.

В системах управления всегда присутствуют независимые события, поэтому в целях упрощения описания, реализации и сопровождения алгоритма активно используется логический параллелизм - каждое независимое событие (или подгруппа событий) обрабатывается отдельным процессом, даже если алгоритм исполняется на одном процессоре. На выполнение процесса требуется некоторое ненулевое процессорное время. Поскольку события возникают независимо, процессы могут «накладываться» друг на друга, интерферировать. Это приводит к неравномерности вычислительной загрузки платформы, появлению режима пиковой загрузки процессора. Оценка ресурсоемкости алгоритма сводится к исследованию пикового режима - основной динамической характеристики системы [1]. Сложность реальных систем не позволяет на практике проводить такие исследования «вручную» - ни аналитически, ни экспериментально [2]. Таким образом, автоматизация исследований динамических характеристик алгоритмов позволяет повысить экономическую эффективность разработок систем управления.

Широко распространенные публикации на тему времени реакции систем, проходящие по направлению т.н. «реального времени», как правило, касаются лишь выделенных алгоритмов планирования в рамках многозадачной модели логического параллелизма. При этом предлагаемые решения не обеспечивают приемлемого уровня универсализма: а) в силу закрытости исходных кодов динамические характеристики выявляются экспериментально [3]; б) не рассматриваются вопросы автоматизации исследований динамических характеристик алгоритмов; в) игнорируются методы распределения вычислительных ресурсов, альтернативные многозадачным, в частности, многопоточные модели организации логического параллелизма.

Между тем, многопоточные модели организации параллелизма в последнее время переживают ренессанс [см., например, 4, где дана обширная библиография по теме]. Этот подход с успехом используется на практике [5]. К нему проявляет пристальный интерес научное сообщество. Особый подкласс таких систем - многопоточная реализация гипер-автомата (МРГА) [5, 6, 7], ориентированная на задачи описания управляющих алгоритмов. С одной стороны, в таких средах возможен априорный анализ динамических характеристик [7], а с другой - существующие лингвистические средства МРГА (язык РЕФЛЕКС, упоминавшийся при описании предыдущих версий под аббревиатурой СПАРМ) можно расширить средствами управления вычислительной загрузкой.

В докладе:

- перечисляются критерии, влияющие на поведение систем логического параллелизма;
- приводится математическая модель поведения системы на базе МРГА;
- определяются средства бесшовного сопряжения исходных текстов с программным обеспечением (ПО) автоматизированного анализа динамических характеристик алгоритма;
- рассматриваются функциональные требования к ПО автоматизированного анализа;

- предлагается расширение синтаксиса/семантики для языка РЕФЛЕКС лингвистическими средствами управления распределением вычислительных ресурсов;
- обсуждаются методы реализации распределения ресурсов в системах на базе МРГА.

2. Краткий обзор проблемы вычисления времени реакции на внешнее событие

Вопрос о том, сколько времени займет вычисление запрограммированного алгоритма, возник одновременно с появлением цифровых систем. Как это ни парадоксально, но по мере развития аппаратной платформы вычислительных систем этот вопрос становится все более запутанным и трудноразрешимым.

Время выполнения заданного алгоритма можно определить тремя способами: с помощью непосредственного измерения времени исполнения программы на целевой системе, методом имитационного моделирования и путем аналитических методов расчета - через исследование исходного текста программ или алгоритмически эквивалентных ему представлений [9].

Метод непосредственного измерения является одним из наиболее распространенных на практике. Суть метода - включение в исполняемый код дополнительных функций мониторинга или организация супервизорного мониторинга, и последующий сбор статистики во время исполнения программы [10, 11]. Он достаточно прост в реализации, однако теоретические исследования этого метода приводят к неутешительным заключениям относительно ценности получаемых данных, которые могут быть подытожены тезисом Дейкстра: «Тестирование программ может служить для доказательства наличия ошибок, но никогда не докажет их отсутствия» [12]. Другим недостатком метода является невозможность его использования на стадии разработки ПО [9].

Метод имитационного моделирования заключается в построении модели вычислительной платформы, что является трудоемкой и нетривиальной задачей. Он позволяет исследовать достаточно широкий класс систем, однако если есть возможность провести априорный расчет, применение этого метода нецелесообразно [9].

Метод априорного расчета основан на анализе исходного текста программы и подсчете времени исполнения инструкций [13, 14]. Однако при очевидной привлекательности он имеет высокую погрешность оценок, не предоставляет универсального решения и тяжело автоматизируется. Среди основных проблем, возникающих при использовании этого метода для реальных программ, можно указать:

- очень большое число возможных путей исполнения алгоритма, требующих анализа;
- трудности с определением длины пути при обработке описаний циклов и рекурсивных подпрограмм;
- недоступность для анализа исходных кодов библиотечных функций [15].

Кроме того, следует отметить обязательную синтаксическую ориентированность средства анализа и, как следствие, привязку к языку описания алгоритма.

Вычислительные системы, для которых производится определение динамических характеристик, традиционно делятся на системы физического параллелизма (многопроцессорные и многомашинные системы с разделяемой памятью, многомашинные системы с распределенной памятью) и системы логического параллелизма. В системах физического параллелизма решается задача организации вычислений на множестве процессоров, а в системах логического параллелизма - проблема описания алгоритма в виде множества взаимосвязанных процессов [16]. Следует добавить, что системы физического параллелизма - это системы, где на первом месте стоит задача повышения вычислительной мощности и вычленения участков кода, допускающих параллельное исполнение. Системы логического параллелизма - это системы, где к проблеме организации эффективного исполнения добавляется задача адекватного представления алгоритма, который содержит участки, допускающие независимую обработку. В связи с этим для систем физического параллелизма характерна проблема статической и динамической балансировки вычислительной нагрузки в однородной или неоднородной среде исполнительных элементов [9, 17].

Среды логического параллелизма подразделяются на системы, построенные на базе операционных систем (ОС) (т.н. системы многозадачного параллелизма), и системы многопоточного параллелизма, где ОС при организации параллелизма не используется. ОС может присутствовать в ПО системы, но играет при этом крайне незначительную роль, например, служит только для загрузки системы. Многопоточный способ организации параллелизма крайне привлекателен из-за компактности представления, низких накладных расходов и простоты отладки. Среди недостатков способа называют проблемы с динамическим созданием новых потоков и необходимость соблюдения дисциплины при их написании [2]. Некоторые недостатки подхода могут быть устранены за счет использования специализированных лингвистических средств [6, 7]. Наиболее подходящая схема организации параллельного вычисления базируется на теории конечных автоматов [18] или семантически близких к конечным автоматам сетях Петри [19]. Несмотря на указанные недостатки, подход широко распространен в области автоматизации [20]. Пример низкоуровневой организации систем такого рода на языке Си можно посмотреть в [21].

Исследование характеристик систем многозадачного логического параллелизма крайне осложнено закрытостью исходных кодов ОС. Неизвестна длина нереентерабельных участков кода ОС, невозможно проанализировать поведение системы при росте числа задач, при одновременном возникновении событий в системе и т.п. Сложные алгоритмы организации планирования задач в системе крайне затрудняют проведение анализа [22]. Иногда для динамических характеристик систем многозадачного логического параллелизма применяются вероятностные описания [23]. Констатируется, что для общего случая многозадачного логического параллелизма решение задач трассировки, определения времени реакции, эффективности и т.п. характеристик программ аналитическими методами или непосредственными исследованиями невозможно [24]. На настоящий момент наиболее подробная информация о динамических характеристиках распространенных ОС представлена в [3]. Информация получена по результатам экспериментальных исследований характеристик ОС. Отчет содержит весьма полезные сведения, однако данных,

позволяющих представить поведение исследованных ОС в формальном виде, в работе так и не получено.

Исследование систем логического многопоточного параллелизма также связано с рядом трудностей. Несмотря на отсутствие неконтролируемого кода ОС, остается проблема аппаратно-обусловленной неоднозначности динамического поведения системы, связанной с кэш-памятью процессора и конвейером [25]. А также проблема недоступных семантико-синтаксическому анализу используемых библиотечных функций и процедур обработки прерываний. Имеются перспективные разработки, включающие средства верификации динамических характеристик ПО, например, исследовательский программный комплекс RT-MEC [26], в котором исследуемые алгоритмы представлены в виде сетей Петри. К сожалению, сети Петри, широко применяемые при теоретических исследованиях, не имеют достаточной степени абстрактности и структурности [27], что несколько сужает область применения комплекса. На применимость данного подхода негативным образом сказывается необходимость преобразования исходных текстов к форме сетей Петри. Вызывает вопросы и отсутствие гарантии того, что трансляция двух различных представлений алгоритма (исходное и в терминах сетей Петри) приведет к идентичному машинному коду.

Резюмируя вышесказанное, нужно отметить, что среди имеющихся на рынке широко распространенных средств разработки ПО нет средств, позволяющих априорно исследовать динамические характеристики создаваемых пользователем алгоритмов. Несомненно, среди основных причин присутствуют и перечисленные сложности, возникающие при реализации этих возможностей.

3. Гипер-автомат и его многопоточная реализация

Существует несколько подходов при организации систем логического многопоточного параллелизма. Один из способов - строить системы на основе модели гипер-автомата [5]. Гипер-автоматом называется совокупность взаимосвязанных *z*-автоматов (процессов) - суть классических конечных С-автоматов с расширенными свойствами, относящимися к групповому взаимодействию процессов [8].

Математической моделью гипер-автомата является множество из трех элементов:

$$H = \{T_H, Z, z_1\},$$

где T_H - *длительность цикла гипер-автомата*; Z - *множество z-автоматов* ($Z = \{z_1, z_2, ..., z_M\}$, где M - число *z*-автоматов гипер-автомата); z_1 - *начальный z-автомат*, $z_1 \in Z$.

Математической моделью *z*-автомата является множество из четырех элементов:

(1) $$z_i = \{S_i, S_i^p, s_i^1, T_i^z\} \quad (z_i \in Z, i = \{1, ..., M\}),$$

где S_i - *множество состояний*; S_i^p - *множество пассивных состояний*, $S_i^p \in S_i$; s_i^1 - *начальное состояние*, $s_i^1 \in S_i \wedge s_i^1 \notin S_i^p$; T_i^z - время отсутствия смены состояния *z*-автомата.

В свою очередь *j*-е состояние произвольного *i*-го *z*-автомата

$$s_i^j = \{X_i^j, Y_i^j\} \quad ,$$

где $X_i^j = \left\{x_i^{j1}, \ldots, x_i^{jL}\right\}$ - *множество событий* состояния; $Y_i^j = \left\{y_i^{j1}, \ldots, y_i^{jL}\right\}$ - *множество реакций* состояния.

Важно отметить, что

$$\forall s_i^j \in S_i^p \Rightarrow X_i^j = \{\varnothing\} \wedge Y_i^j = \{\varnothing\} ,$$

т.е. пассивные состояния характеризуются тем, что их множества событий и множества реакций пусты, т.е. z-автомат, попав в одно из своих пассивных состояний, *самостоятельно* не может изменить его на другое. Среди пассивных состояний выделяют s_i^{ns} - состояние «нормального останова» - и s_i^{es} - состояние «останова по ошибке».

В течение одного цикла гипер-автомата z-автомат может находиться в одном и только одном состоянии и, соответственно, может сформировать за один цикл одну и только одну реакцию. Текущее состояние *i*-го *z*-автомата формально обозначается как s_i^{cur}.

Начальный *z*-автомат z_1 характеризуется тем, что по инициализации системы он находится в своем начальном состоянии (s_1^1). Все остальные *z*-автоматы ($z_i \in Z, i = \{2, \ldots, N\}$) находятся в выделенных пассивных состояниях s_i^{ns}.

Если *z*-автомат находится в состоянии, *не* принадлежащем множеству пассивных состояний, то говорят, что *z*-автомат активен (находится в активном состоянии). В противном случае говорят, что *z*-автомат пассивен (находится в пассивном состоянии).

В качестве *события* может рассматриваться произвольная суперпозиция фактов: нахождение некоторого *z*-автомата в некотором состоянии, некоторое определенное значение переменной и некоторое определенное значение времени отсутствия переходов *z*-автомата.

Реакцией называется произвольная суперпозиция действий по изменению значений переменных и состояний *z*-автоматов.

Различаются *входные переменные*, значения которых формируются вне гипер-автомата и используются внутри гипер-автомата; *выходные переменные*, значения которых формируются внутри гипер-автомата и используются вне гипер-автомата; и *внутренние переменные*, которые и формируются и используются только внутри гипер-автомата.

Модель явным образом предполагает цикличность исполнения гипер-автомата, что снимает психологические трудности, возникающие при работе с классическими моделями конечного автомата [28]. Модель гипер-автомата предполагает существование внешней по отношению к гипер-автомату среды, что позволяет использовать эту модель для создания гомеостатических систем, например, систем управления.

Многопоточная реализация гипер-автомата (язык РЕФЛЕКС) является специализированным языком программирования систем логического параллелизма, ориентированным на описание алгоритмов функционирования сложных автоматизированных или полностью автоматических систем. Язык выполнен в виде проблемно-ориентированного расширения языка Си. Текущая версия языка предполагает несколько упрощеный вариант гипер-автомата:

- множество конечных состояний содержит только два элемента - состояние «останова по ошибке» и состояние «нормального останова»;

- текущий *z*-автомат (или процесс) может произвести полный набор возможных реакций изменения состояний только в отношении самого себя. Для остальных процессов допустим только ограниченный набор операций по изменению состояний: запуск с начального состояния, перевод в состояние «нормального останова» и перевод в состояние «останова по ошибке».

Алгоритм работы некоторого дискретного устройства в языке РЕФЛЕКС описывается в текстовом виде как совокупность слабосвязанных процессов. Процессы исполняются параллельно. Извне процессы можно останавливать и запускать сначала. Из каждого процесса можно получить информацию о состоянии любого другого процесса. Процессы равноправны и не связаны «родственными» отношениями.

Указанные свойства позволяют организовывать процессы гипер-автомата в иерархические структуры произвольного порядка (в том числе и замкнутые). Языковая среда располагает к использованию событийно-управляемой модели для описания алгоритма. Это, в свою очередь, позволяет применять любую из существующих стратегий управления [29]. Исследование базовых характеристик языка на соответствие требованиям надежности приводит к заключению о допустимости его применения в задачах автоматизации сложных объектов критичных производств [7].

РЕФЛЕКС имеет текстовый русскоязычный синтаксис, что облегчает работу с ним отечественных пользователей, и синтаксически базируется на языке Си, что упрощает его изучение большинством практикующих программистов. Эквивалентное отображение описания алгоритма в машинный код также базируется на языке Си. В силу специфических особенностей РЕФЛЕКС (параллелизм, русскоязычный синтаксис и полноценная идентификационная система, модифицированная типизация переменных и необходимость контроля этой типизации, несколько отличная от Си семантика, информационная изоляция пользователя от деталей реализации) алгоритмически эквивалентное преобразование в язык Си осуществляется трансляционной моделью. Переносимость обеспечивается выделением аппаратно-зависимых процедур [6]. Простая и эффективная организация логического параллелизма не исключает использование языка в средах физического параллелизма (см. например [5], где описано применение языка для построения многопроцессорной системы на базе магистрали Multibus).

4. Расширение синтаксиса языка РЕФЛЕКС

Для предыдущих версий языка РЕФЛЕКС были показаны наличие свойства временной предсказуемости и существование заведомо конечного времени реакции системы управления на внешнее событие, что обусловлено способом реализации многопоточного логического параллелизма [7]. В текущей версии языка были введены конструкции многовариантного переключения и облегченный механизм интеграции с Си-библиотеками. Таким образом, если конструкции многовариантного переключения оставляют полученные раннее результаты без изменения, то возможность использования библиотечных функций несколько усложняют задачу определения времени реакции системы. Кроме этого, полученные ранее результаты не учитывали принципиальную

неопределенность времени исполнения, вносимую такими аппаратными решениями, как кэш-память и конвейер [25].

Во время длительной эксплуатации языка постоянно возникают предложения по коррекции и расширению синтаксиса. На настоящий момент можно констатировать, что базовый синтаксис языка устоялся. Выдвинут основной критерий дальнейших модификаций - сохранение прозрачности базовой концепции и совместимость с предыдущими версиями. Расширение синтаксически допустимых конструкций идет в части оптимизации работы программиста. Это разрешение сокращенных форм операций в контекстно однозначных случаях и введение незначительного числа дополнительных терминальных символов, которые существенно повышают модифицируемость программ.

В настоящее время активно обсуждается вопрос введения средств задания для *z*-автоматов индивидуальных делителей базовой тактовой частоты гипер-автомата, что позволит производить балансировку вычислительной загрузки за счет более гибкой подстройки времени отклика системы на внешние события. Существующие решения по индивидуальной подстройке времени отклика предполагает серию рутинных операции, упрощение которых способно повысить надежность средства программирования.

Введение этих новых свойств несколько меняет математическую модель гипер-автомата. Теперь нужно в определение *z*-автомата (1) ввести понятие индивидуального делителя частоты:

$$z_i = \{S_i, S_i^p, s_i^1, T_i^z, D_i, O_i\} \ \ (z_i \in Z, i = \{1,\ldots, M\}),$$

где S_i - множество состояний; S_i^p - множество пассивных состояний, $S_i^f \in S_i$; s_i^1 - начальное состояние, $s_i^1 \in S_i \wedge s_i^1 \notin S_i^p$; T_i^z - время отсутствия смены состояния *z*-автомата; D_i - индивидуальный делитель частоты гипер-автомата, $D_i \in \aleph$; O_i - индивидуальное смещение, $O_i \in \{0,1,\ldots, D_i - 1\}$.

Индивидуальный делитель частоты устанавливает для *i*-го *z*-автомата длительность цикла, равную $D_i * T_H$, что эквивалентно соответствующему увеличению времени реакции *z*-автомата на событие. При этом сохраняется преемственность с первоначальным определением гипер-автомата, как гипер-автомата, состоящего только из *z*-автоматов, имеющих индивидуальный делитель частоты, равный единице. Индивидуальное смещение определяет, в какой из тактов будет исполнено очередное состояние *z*-автомата. Выполнение очередного состояние *i*-го *z*-автомата происходит тогда и только тогда, когда остаток от деления числа прошедших с момента запуска гипер-автомата тактов на D_i равен O_i. Ожидаемый выигрыш от введения нового свойства - возможность балансировки вычислительной нагрузки.

На языковом уровне введение нового свойства выражается в появлении дополнительной возможности установить индивидуальную тактовую частоту процесса. Это делается посредством ключевого слова `ДЕЛИТЕЛЬ` с указанием числа.

Например:

```
ПРОЦ КонтрольПерегреваТоковводов ДЕЛИТЕЛЬ 10 {
<тело процесса>
}
```

т.е. для одного из процессов, осуществляющих мониторинг температуры некоторого объекта автоматизации, время реакции на внешнее событие увеличено в десять раз по сравнению с базовым временем реакции гипер-автомата T_H.

Другой активно обсуждаемый вопрос по расширению синтаксиса касается введения понятия группы конкурирующих *z*-автоматов ($G_i = \{z_i^1, \ldots, z_i^k\}$). Конкурирующие *z*-автоматы - это процессы, одновременное исполнение которых может привести к противоречию. Группа конкурирующих *z*-автоматов является подмножеством Z ($G_i \in Z$). Группы конкурирующих *z*-автоматов не пересекаются:

$$\forall G_i \in Z \wedge \forall G_j \in Z : (i \neq j) \quad G_i \cap G_j = \{\varnothing\}.$$

Все члены группы конкурирующих *z*-автоматов имеют одинаковый делитель частоты. Вырожденная группа - это группа, содержащая единственный элемент. При этом сохраняется преемственность с первоначальным определением гипер-автомата, как гипер-автомата, состоящего только из вырожденных групп. Основное свойство группы конкурирующих *z*-автоматов - если один из *z*-автоматов группы активен, значит, все остальные *z*-автоматы группы пассивны. Вопрос о введении понятия группы конкурирующих *z*-автоматов возник после того, как «гипер-автоматное» описание алгоритмов работы реальных объектов автоматики показало очень сильные структурирующие свойства. При этом зафиксировано значительное снижение сложности описания алгоритма за счет того, что он представляется в виде компактных и информационно изолированных друг от друга *z*-автоматов. Математически это упрощение выражается в том, что алгоритмически эквивалентный классический конечный автомат должен иметь число состояний, равное произведению числа состояний независимо работающих *z*-автоматов. Однако для многомерных алгоритмов это снижение сложности означает большое число *z*-автоматов (в последнем проекте число процессов приближается к шестистам, число состояний алгоритмически эквивалентного конечного автомата порядка 10^{30} (sic!)). Ожидаемый выигрыш от введения понятия группы конкурирующих *z*-автоматов - возможность автоматического разрешения конфликтных ситуаций и повышение надежности создаваемых программ.

5. Балансировка вычислительной нагрузки и расчет времени реакции на внешнее событие

Для детерминированных систем, где механизмы своппинга, кэш-памяти и конвейеризации не вносят динамической неопределенности во время исполнения, формальное описание динамических характеристик МРГА получено в [7]. Общая методика вычислений подобна описанной в [14]. Максимальное гарантированное время реакции системы на внешнее событие ($T_{макс}$) определяется временем $T_{накл}$, затраченным на накладные расходы по организации функционирования алгоритма (связь с физическими устройствами ввода/вывода, организация параллельного исполнения), и временем, затраченным собственно на выполнение алгоритма ($T_{алг}$):

$$T_{макс} = 2\frac{(T_{накл} + T_{алг})}{(1-K)},$$

где $K \in [0,1]$ - доля вычислительной мощности системы, затраченная на обработку прерываний при максимально допустимой частоте их поступления (K = 1 означает, что выбранная целевая система будет заниматься только обработкой прерываний).

$T_{накл}$ определяется как сумма времени, затраченного на организацию параллелизма, и времени, затраченного на обслуживание ввода/вывода. Обслуживание ввода/вывода включает в себя:

- время на считывание физических портов;
- время на преобразование считанных значений во внутренние программные переменные;
- время на обратное преобразование внутренних переменных к виду, пригодному для выдачи в порт;
- время выдачи в выходные порты [7].

Если значение $T_{накл}$ не вызывает проблем с вычислением, то с введением понятий групп, делителей, с расширением синтаксиса операторами многовариантного перехода вычисление $T_{алг}$ несколько изменяется. Однако по-прежнему остаются два ключевых принципа:

- оценка ведется по пиковой нагрузке гипер-автомата;
- при расчетах *z*-автомат характеризуется временем исполнения своего наиболее ресурсопотребляющего (пикового) состояния.

Поиск пиковой загрузки системы включает следующие этапы:

- определение пиковых состояний *z*-автоматов;
- статическую балансировку вычислительной нагрузки;
- определение интерферирующих (параллельно исполняемых) элементов на основании информации о взаимоисключающих *z*-автоматах;
- определение максимальных времен исполнения интерферирующих элементов и нахождение искомого $T_{алг}$ как суммы найденных времен.

5.1. Определение пиковых состояний *z*-автомата

Нахождение времени исполнения состояния *z*-автомата заключается в определении наиболее ресурсопотребляющего пути исполнения алгоритма и подсчете времени исполнения инструкций этого пути [13, 14]. С введением операторов многовариантного разбора поиск такого пути усложнен появлением древовидной структуры пути. Однако, в силу конструктивной невозможности организации внутри состояния обратного течения управления, древовидная структура возможных путей линеаризуется и может быть проанализирована во время одного из штатных проходов транслятора языка.

Например, для текста:

```
ЕСЛИ (событие) {
      РАЗБОР (событие){
            СЛУЧАЙ <константа>:
                  <линейные инструкции>
                  КОНЕЦ;
            СЛУЧАЙ <константа>:
                  <линейные инструкции>
                  КОНЕЦ;
            УМОЛЧАНИЕ:
```

```
                    <линейные инструкции>
                    КОНЕЦ;
        }
} ИНАЧЕ {
        ЕСЛИ (событие) {
                <линейные инструкции>
        } ИНАЧЕ {
                <линейные инструкции>
        }
}
```

процедура линеаризации разбора будет выглядеть так, как показано на рис.1.

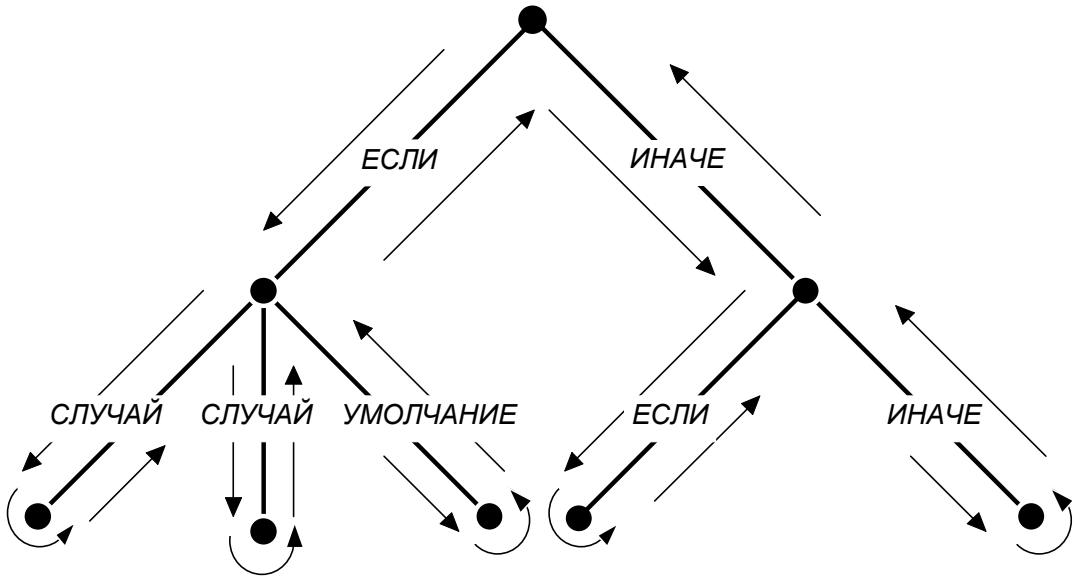

Рис.1. Линеаризация обхода возможных путей исполнения алгоритма. В узлах происходит сравнение насчитанного времени исполнения и определение максимально ресурсопотребляющего пути

Пиковым состоянием алгоритма является состояние с максимальным временем исполнения. Характеристическое время *z*-автомата, необходимое для дальнейших вычислений, равно времени исполнения пикового состояния. Характеристическое время *i*-го *z*-автомата обозначается $Max(z_i)$. Трудности, возникающие при подсчете этого времени:

- необходимость различать времена исполнения т.н. «ad hoc» полиморфных операций Си-ориентированного синтаксиса;
- отсутствие априорных данных о времени исполнения библиотечных функций;
- динамическая недетерминированность времени исполнения инструкций, вызванная кэш-памятью и конвейеризацией.

5.2. Статическая балансировка вычислительной нагрузки и расчет пиковой нагрузки *z*-автомата

Статическая балансировка осуществляется автоматически путем присвоения индивидуального смещения каждому из *z*-автоматов, имеющих делитель смещения, отличный от единицы. Для членов одной группы

конкурирующих *z*-автоматов устанавливается единое смещение. Присвоение индивидуальных смещений производится по следующему алгоритму:

- внутри групп конкурирующих *z*-автоматов определяется характеристическое время исполнения группы $Max(G_j) = Max(Max(z_i))$ по всем $z_i \in G_j$. Для вырожденных групп $Max(G_k) = Max(z_l)$ т.к. член группы только один;
- группы объединяются по признаку равенства индивидуальных делителей частоты и в дальнейшем рассматриваются по объединениям, независимо;
- внутри объединения группы ранжируются по характеристическим временам;
- для каждого из индивидуальных смещений, возможных для рассматриваемого делителя, заводится счетчик суммарного времени $T_i^{sum}, i \in \{0,\ldots,D_i\}$;
- распределение индивидуальных смещений производится по группам. По порядку, начиная с группы с наибольшим характеристическим временем. Элементам рассматриваемой группы присваиваются индивидуальное смещение, равное номеру *i* счетчика суммарного времени с минимальным значением $(O_j = i : T_i^{sum} = Min(T_k^{sum}))$. После чего счетчик суммарного времени увеличивается на величину $Max(G_k)$.

Очевидно, что предложенный алгоритм не гарантирует оптимального решения. Однако предлагает разумный компромисс между простотой и качеством решения для большого числа однородных *z*-автоматов.

Другой недостаток предложенного алгоритма - отсутствие учета возникновения в системе эффекта кратности. Процессы с кратными индивидуальными делителями будут образовывать устойчивые структуры. Например, процессы из объединения с индивидуальным делителем 4 и индивидуальным смещением 0 будут всегда исполняться с процессами, имеющими индивидуальный делитель 2 и смещение 0.

При этих условиях нахождение оптимального решения, скорее всего, означает проведение полного комбинаторного перебора, что для реального числа процессов (порядка тысячи) является на настоящий момент неразрешимой задачей. Например, для сотни *z*-автоматов и индивидуального делителя частоты равного двум общее число возможных вариантов равно

$$\frac{1}{2}\sum_{i=0}^{100} C_{100}^{i} = \frac{1}{2}\sum_{i=0}^{100} \frac{100!}{i!(100-i)!},$$

что по самым грубым оценкам в предположении, что один вариант обсчитывается за одну микросекунду, требует времени заведомо большего 10^9 секунд или порядка тридцати лет.

После проведения балансировки нагрузки для каждого индивидуального делителя нужно выявить счетчики характеристического времени, содержащие максимальную величину, затем выявленные значения просуммировать. Полученное число является верхней оценкой $T_{алг}$.

Ввиду того, что $T_{алг}$ определено для случая выполнения всеми *z*-автоматами своих наиболее ресурсопотребляющих состояний, реально наблюдаемое время реакции системы будет равно

$$T_{\text{реал}} \in (T_{накл}, T_{макс}).$$

В общем случае на базе возникшей алгоритмической ситуации можно сформулировать следующую задачу.

Задача о горошинах. Дано n одинаковых стаканов и m горошин разной априорно известной массы. Требуется разложить горошины по стаканам так, чтобы разница масс стаканов с горошинами была минимальна.

6. Функциональные требования к программному обеспечению автоматизированного анализа

ПО автоматизированного анализа должно удовлетворять следующим минимальным требованиям:

- предоставлять необходимые тексты на языке Си программы автоматического замера времени исполнения базовых языковых конструкций и служебных подпрограмм;
- предоставлять необходимые тексты-шаблоны на языке Си программы автоматического замера времени исполнения процедур обработки прерываний и расчета доли вычислительной мощности, затрачиваемой на обработку прерываний;
- автоматически генерировать тексты-шаблоны на языке Си программ замера времени исполнения библиотечных функций;
- формировать базы данных временных характеристик вычислительных платформ, содержащие информацию о временах исполнения базовых языковых конструкций, служебных подпрограмм и библиотечных функциях;
- на основании информации из базы данных временных характеристик вычислительных платформ автоматически анализировать исходный текст программы на языке РЕФЛЕКС и определять пиковую загрузку процесса в режиме отсутствия прерываний;
- на основании информации о доле вычислительной мощности, затрачиваемой на обработку прерываний, и значении времени, определенном на предыдущем шаге, делать оценку возможности реализации желаемой тактовой частоты гипер-автомата на заданной платформе, а также делать прогноз времени реакции на внешнее событие для z-автоматов с различными делителями тактовой частоты гипер-автомата.

Тексты вспомогательных программ для замера времен исполнения должны компилироваться штатным компилятором Си и запускаться на исполнение целевой платформе. В результате исполнения программа должна генерировать файл временных характеристик вычислительной платформы в стандартизированном формате. Общий алгоритм определения временных характеристик для базовых языковых конструкций, служебных подпрограмм и библиотечных функций совпадает и показан на рис. 2.

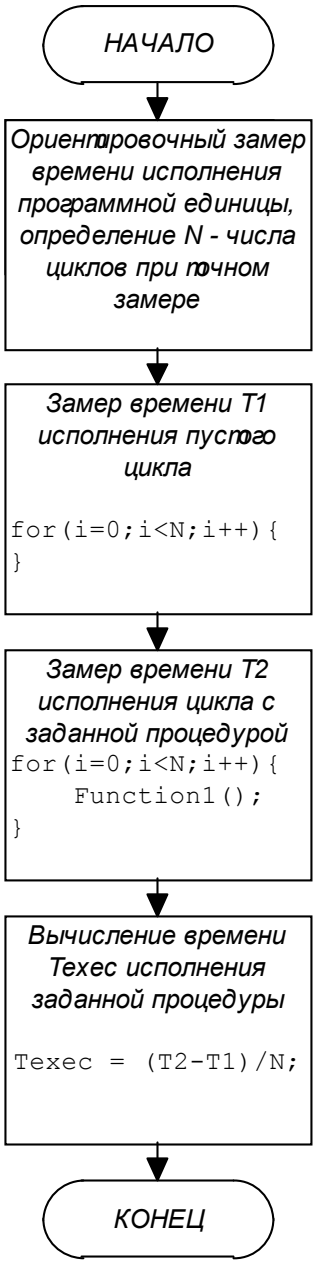

Рис. 2. Блок-схема вычисления времени исполнения заданной процедуры

Поскольку автоматическое определение времени исполнения библиотечных процедур невозможно, генерируются тексты-шаблоны, в которых пользователь должен установить значения параметров, соответствующие наиболее ресурсоемким путям выполнения.

Та же методика применима и для процедур обработки прерываний. Однако дополнительно пользователь должен указать максимальную частоту следования прерываний. Тогда на основании полученных замеров времени и максимальной частоты прерываний, заданных пользователем, вычисляется коэффициент K (доля загрузки процессора на обработку прерываний):

$$K = \sum_{i=1}^{N} f_i \tau_i ,$$

где N - число источников прерываний в системе; f_i - частота следования прерываний от i-го источника, Гц; τ_i - время обработки прерываний от i-го источника, сек.

Поскольку разбор исходного текста программы линеаризуется, определение пиковых состояний z-автоматов может быть совмещено с этапом синтаксического анализа. Определение пиковых состояний групп конкурирующих процессов и распределение смещений требуют отдельного прохода, который может быть совмещен с этапом семантического анализа текста программы и кодогенерации выходного Си-файла существующим двухпроходным транслятором языка РЕФЛЕКС.

7. Обсуждение результатов

Краткий обзор существующего положения в области подходов исследования динамических характеристик систем показывает, что до настоящего времени так и не предложено удовлетворительного решения проблемы. Косвенно этот факт находит отражение в том, что при чрезвычайно частом использовании термина «реальное время» в профессиональной (и около-профессиональной литературе) на рынке отсутствуют средства априорного исследования динамических свойств алгоритмов. По результатам исследований можно констатировать, что априорное определение пиковых нагрузок системы связано с рядом принципиальных трудностей, не позволяющих сделать точную оценку. Среди основных источников погрешности нужно указать:

- трудности с интеграцией разбора «ad hoc» полиморфизма в существующую версию транслятора, в которой этот разбор перекладывается на уровень вторичного целевого компилятора;
- априорно неизвестные оптимизирующие свойства целевого компилятора Си;
- аппаратно-обусловленная неопределенность при использовании кэш-памяти, конвейеризации и процессоров с суперскалярной архитектурой;
- необходимость ручной настройки при измерении времен исполнения библиотечных функций и процедур обработки прерываний.

Разбор «ad hoc» полиморфизма требует доработки алгоритмов синтаксического разбора перед их использованием в ПО анализа. Неопределенность с оптимизирующими свойствами целевого компилятора устраняется, если обеспечить идентичность ключей компиляции для текстов вспомогательных программ замера времени исполнения и реальных управляющих алгоритмов. Аппаратно обусловленная неопределенность может приводить к погрешности оценки времен исполнения, равную нескольким десяткам процентов [30], и является принципиально неустранимой. Ожидается снижение влияния, по крайней мере, кэш-памяти за счет чрезвычайно компактного исполняемого кода, что обусловлено одним из свойств применяемой методики. Погрешность, обусловленную конвейеризацией и суперскалярностью, можно попытаться снизить за счет использования статистических методов, путем использования поправочных коэффициентов (что, впрочем, требует отдельного рассмотрения, подкрепленного серией

экспериментальных работ). Необходимость ручной настройки некоторых параметров является потенциальным источником погрешностей вычисления, обусловленных в первую очередь человеческим фактором.

Затраты на создание ПО анализа требуют:

- коррекции алгоритмов трансляции с учетом расширения синтаксиса;
- реализации алгоритмов определения пиковых состояний *z*-автоматов, в частности, введение разбора случаев «ad hoc» полиморфизма;
- разработку структуры баз данных для различных аппаратных платформ.

В целом описанная методика и использованные средства обеспечивают бесшовное сопряжение исходных текстов с ПО анализа, не накладывают дополнительных ограничений на способ формирования исходного текста, предполагают типизированную последовательность операций и накопление баз данных.

8. Заключение

В работе была исследована возможность расширения языка РЕФЛЕКС, являющегося многопоточной реализацией гипер-автоматной модели, лингвистическими средствами управления распределением вычислительных ресурсов. Показано, что синтаксис языка РЕФЛЕКС допускает расширение средствами управления вычислительными ресурсами системы, которые не противоречат его семантике. Обсуждены проблемы априорного определения времени исполнения процессов, необходимого для качественной балансировки нагрузки. Указаны компоненты, определяющие время исполнения и частные способы их определения. Предложен механизм индивидуального делителя частоты, позволяющий независимо регулировать время реакции на отдельное внешнее событие, и субоптимальный алгоритм статической балансировки вычислительной мощности. Выявлены основные источники погрешности определения времени реакции. Перечислены основные функциональные требования к программному обеспечению априорного анализа. Рассмотренные аспекты проблемы позволяют сделать заключение о допустимости и полезности предлагаемого расширения синтаксиса языка РЕФЛЕКС, а также сделать заключение о возможности создания программного обеспечения априорного анализа динамических характеристик алгоритмов, совмещающего удовлетворительное качество получаемых оценок с приемлемыми расходами на реализацию и бесшовным сопряжением с исходным текстом. Разработанные принципы организации управляющих алгоритмов позволяют вывести проектирование систем управления, построенных на базе МРГА, на качественно новый уровень: изменять и априорно исследовать временные характеристики алгоритма.

Список литературы

1. Liu C. L., James W. Layland Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment // Journal of the Association for Computing Machinery. 1973. Vol. 20, No 1. P. 46-61.
2. Сорокин С. Системы реального времени // Современные технологии автоматизации. 1997. №2. С. 22-29.
3. Dedicated Systems Experts. RTOS evaluation program. 2002. http://www.dedicated-systems.com.
4. Легалов.А.И. Разноликое программирование. http://www.softcraft.ru.
5. Зюбин В.Е. Проектирование алгоритмов работы микроконтроллеров // Приборы и системы управления. 1998. №1. С. 18-20.
6. Зюбин В.Е. Язык СПАРМ - средство программирования микроконтроллеров // Автометрия. 1996. №2. С. 40-50.
7. Зюбин В.Е. Исследование условий применимости языка параллельного программирования СПАРМ для задач построения надежных управляющих программ // Распределенная обработка информации: Тр. / Шестой международный семинар. -Новосибирск. 1998. С. 122-126.
8. Зюбин В.Е. Средство программирования алгоритмов работы микроконтроллеров - СПАРМ // Распределенная обработка информации: Тр. / Пятый международный семинар. Новосибирск, 1995. С. 86-91.
9. Головкин Б.А. Расчет характеристик и планирование параллельных вычислительных процессов. М.: Радио и связь, 1983. 273 с.
10. Куцевалов А.В., Куцевалов Д.В. Организация процесса временного анализа программ // Программирование. 1978. №4. С. 44-47.
11. Капырин В.А., Халецкий А.К. Способ оценки надежности функционирования программ реального времени // Программирование. 1982. №3. С. 73-79.
12. Гордиенко А.В. Тестирование при оценке динамической корректности программ АСУ // Программирование. 1982. №6. С. 48-52.
13. Липаев В.В., Серебровский А.Л., Филиппович В.В. Принципы построения и основные требования к системам автоматизации программирования и отладки программ для управляющих систем // Программирование. 1975. №2. С. 55-60.
14. Данильченко Л.С., Людвиченко В.А. О получении априорных оценок времени решения задач программами на Фортране // Программирование. 1988. №5. С. 56-60.
15. Борзов Ю.В. Тестирование программ с использованием символического выполнения // Программирование. 1980. №1. С. 51-59.
16. Banatre J.B., Routeau J.P. A Multiset Transformation // Communications of the ACM. 1993. Vol. 36. No 1. P. 98-111.
17. Аветисян А.И., Гайсарян С.С., Самоваров О.И. Возможности оптимального выполнения параллельных программ, содержащих простые и итерированные циклы, на однородных параллельных вычислительных системах с распределенной памятью // Программирование. 2002. №1. С. 38-54.
18. Баранов С.И. Синтез микропрограммных автоматов. Л.: Энергия, 1974. 216 с.
19. Котов В.Е. Сети Петри. М.: Наука, 1984. 160 с.
20. IEC 65B/373/CD, Committee Draft - IEC 61131-3. Programmable controllers. Part 3: Programming languages, 2nd Ed. // International Electrotechnic Commission. 1998.
21. Шалыто А.А., Туккель Н.И. SWITCH-технология - автоматный подход к созданию программного обеспечения «реактивных» систем // Программирование. 2001. №5. С. 45-62.
22. Димитров А.А., Каган Б.М., Крейнин А.Я. Аналитические модели вычислительных систем реального времени с фоновыми задачами // Программирование. 1977. №6. С. 60-65.
23. Вайнштейн А.Д., Кадушин А.И. Оценка взаимного влияния задач при пакетной обработке в мультипрограммном режиме // Программирование. 1982. №5. С. 72-79.
24. Кринницкий Н.А., Чернова Т.Ф. Об имитационном моделировании операционных систем // Программирование. 1981. №3. С. 77-85.
25. Балашов В.В., Капитонова А.П., Костенко В.А., Смелянский Р.Л., Ющенко Н.В. Метод и средства оценки времени выполнения оптимизированных программ // Программирование. 1999. №5. С. 52-61.

26. Вербицкайте И.Б., Быстров А.В. Автоматический анализ и верификация распределенных систем реального времени // Распределенная обработка информации: Тр. / Шестой международный семинар. Новосибирск, 1998. С. 236-240.
27. Olderog E.-R. Operational Petri Net Semantics for CCSP // Lecture Notes in Computer Sciences. 1984. Vol. 266. Springer-Verlag.
28. Кузнецов Б.П. Психология автоматного программирования // BYTE. 2000, №11. http://www.softcraft.ru/design/ap/ap01.shtml.
29. Christensen J. IEC 61499 (Function Blocks for industrial-process measurement and control systems) FAQ. 1999. http://www.holobloc.com/stds/iec/tc65wg6/html/faq.htm.
30. Кузьминский М. Архитектурные особенности микропроцессоров PA-8000/8200/8500 // Открытые системы. №3. 1997. С. 6-9. http://www.csu.ac.ru/osp/os/1997/03/source/6.html.