УДК 681.3.06

В. Е. Зюбин

Институт автоматики и электрометрии СО РАН
пр. Акад. Коптюга, 1, Новосибирск, 630090, Россия

Новосибирский государственный университет
ул. Пирогова, 2, Новосибирск, 630090, Россия

E-mail: zyubin@iae.nsk.su

СТАТИЧЕСКАЯ БАЛАНСИРОВКА ВЫЧИСЛИТЕЛЬНОЙ НАГРУЗКИ В ПРОЦЕСС-ОРИЕНТИРОВАННОМ ПРОГРАММИРОВАНИИ ПРИ МНОГОПОТОЧНОЙ РЕАЛИЗАЦИИ

Рассматривается проблема балансировки вычислительной нагрузки при многопоточной реализации процесс-ориентированной программной модели. Предлагается способ гибкой настройки времени реакции процессов на внешнее событие в рамках многопоточной реализации. Представлены бесшовное расширение синтаксиса языка Рефлекс средствами балансировки, метод априорного вычисления времени реакции системы на внешнее событие и субоптимальный алгоритм статической балансировки вычислительной нагрузки.

Ключевые слова: процесс-ориентированное программирование, алгоритмы управления, балансировка вычислительной нагрузки при многопоточной реализации алгоритма, настройка времени реакции системы на внешнее событие, оценка динамических характеристик алгоритма.

Введение

Один из важных вопросов, который возникает при проектировании цифровой системы управления, – вопрос адекватности вычислительной платформы и запрограммированного алгоритма. Использование вычислительной платформы с избыточной мощностью означает рост бюджета проекта, а недостаток мощности означает невозможность обеспечить требуемое качество управления – приемлемое время реакции на внешнее событие.

Несмотря на важность вопроса, общепринятый на практике подход – выбирать платформу «на глазок». Такая плачевная ситуация обусловлена сложностью анализа динамических характеристик программно-аппаратных систем, в первую очередь режима пиковой загрузки – главного предмета исследований при оценке ресурсоемкости алгоритма [1].

Сложность вызвана главным образом низким уровнем структуризации управляющего алгоритма, в большинстве случаев создаваемого средствами языков МЭК 61131-3, что не позволяет анализировать исходные тексты программы на предмет оценки их поведения во время исполнения. Поэтому исследователи констатируют невозможность на практике проводить оценки ресурсоемкости алгоритма ни аналитически, ни экспериментально [2].

Типичные публикации на тему времени реакции системы, проходящие по направлению так называемого «реального времени», касаются лишь выделенных алгоритмов планирования в рамках многозадачной модели логического параллелизма и времени переключения контекста задач [1]. Вопрос о динамическом поведении алгоритма в целом в таких исследованиях даже не ставится.

[1] Dedicated Systems Experts. RTOS evaluation program. 2002. URL: http://www.dedicated-systems.com.

Апостериорные методы – тестирование программы во время исполнения – также не позволяют ответить на вопрос, что же мы получили в результате. Поскольку реакция управляющего алгоритма зависит не просто от состояния входных сигналов, а от истории их изменения (именно поэтому часто говорят о поведенческом характере управляющих алгоритмов и наличии у них памяти), то полное тестирование для сколько-нибудь сложных алгоритмов практически нереализуемо.

В отличие от языков МЭК 61131-3 процесс-ориентированный подход, предполагающий описание алгоритма в виде совокупности взаимодействующих процессов, обеспечивает дополнительный уровень структуризации, позволяющий не только упростить анализ ресурсоемкости алгоритма, но и ввести механизмы балансировки вычислительной нагрузки.

В статье описываются проблемы, связанные с вычислением времени реакции системы на внешнее событие, приводятся основные идеи, заложенные при разработке процесс-ориентированного языка Рефлекс, и вариант многопоточной реализации, предлагается бесшовное расширение синтаксиса языка Рефлекс на основе делителя частоты активизации, позволяющее специфицировать время реакции отдельных процессов, и обсуждаются варианты программно-алгоритмической поддержки предлагаемого нововведения, позволяющие не только оценить, но и снизить общую ресурсоемкость алгоритма.

Проблемы вычисления времени реакции на внешнее событие

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

За последние тридцать лет исследователями было разработано большое число формализмов, ориентированных на создание управляющих алгоритмов. Не претендуя воспроизвести полный список, в качестве примеров можно упомянуть: модель взаимодействующих последовательных процессов (Ч. Хоар) [3], базирующийся на этой модели язык occam [1], диаграммы состояний (Д. Харел) [4], язык Esterel [5], исчисление общающихся систем (Р. Милнер) и его расширения [6–8]. Попытка решить проблему определения времени реакции системы на внешнее событие была предпринята в языке PAISLey [11], однако по заключению самих авторов полученное решение предполагает «нереалистичные ограничения для разработчика» и не может быть названо практически ценным. В рамках проекта Esterel недавно были получены решения [10–13] для проверки гипотезы идеального синхронизма, базирующиеся на идее преобразования Esterel-программы к конечно-автоматному виду и алгоритме обработки, аналогичному предложенному ранее для процесс-ориентированного языка Рефлекс [14]. Вопросы снижения времени реакции на внешнее событие (или в исходной постановке наихудшего времени исполнения «worst-case execution time» WCET) при этом не рассматриваются. Интересный результат, предполагающий редукцию WCET, представлен в [15] для языка Си. При этом остаются за рамками рассмотрения вопросы параллелизма потока управления и влияние прерываний.

Если в конце 1970-х – 1980-х гг. наблюдалось большое число публикаций, связанных с вопросами так называемого «реального» времени, то к началу нулевых энтузиазм исследователей заметно иссяк, и термин «реальное время» стал широко использоваться маркетологами для привлечения внимания потребителя к своим продуктам. Некоторая активизация исследований в этом направлении приходится на конец нулевых, что, по-видимому, связано с технологическим барьером на рост производительности в рамках одноядерной архитектуры, с которым IT-индустрия столкнулась в начале века.

[1] Occam 2.1 REFERENCE MANUAL // SGS-THOMSON Microelectronics Limited 1995. URL: http://www.wotug.org/occam/documentation/oc21refman.pdf (дата обращения: 05.01.2012).

На настоящий момент существует три основных способа определить динамические характеристики заданного алгоритма: 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 [16].

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 [17; 18]. Метод прост в реализации, однако ценность получаемых данных низка в силу тезиса Дейкстры: «Тестирование программ может служить для доказательства наличия ошибок, но никогда не докажет их отсутствия» [19]. Другой недостаток метода – невозможность его использования на стадии разработки ПО [16].

Метод имитационного моделирования заключается в построении модели вычислительной платформы, что представляет трудоемкую и нетривиальную задачу и ограничивает практическое использование метода [16]. Пример реализации такого подхода – система Протеус [20], предоставляющая возможность имитации работы с вычислительными платформами PIC и AVR.

Метод априорного расчета основан на анализе исходного текста программы и подсчете времени исполнения инструкций [21; 22]. Однако при очевидной привлекательности он имеет высокую погрешность оценок, не предоставляет универсального решения и тяжело автоматизируется. Сложность использования метода для реальных программ обусловлена:

- очень большим числом возможных путей исполнения алгоритма, требующих анализа;
- трудностями с определением длины пути при обработке описаний циклов и рекурсивных подпрограмм;
- недоступностью для анализа исходных кодов библиотечных функций [23].

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

Подходы к решению задачи зависят и от способа логической организации ПО систем управления на целевой платформе, которые традиционно делят на системы *многозадачного* и *многопоточного* параллелизма, т. е. с логическим параллелизмом, реализуемым либо на уровне операционной системы (многозадачный логический параллелизм), либо внутри одной задачи (многопоточный логический параллелизм).

Исследование характеристик систем *многозадачного* логического параллелизма 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. Системы многозадачного логического параллелизма предоставляют возможность балансировки вычислительной нагрузки через установку приоритетов. Однако сложные алгоритмы организации планирования задач в системе затрудняют проведение анализа [24]. 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 [25].

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 [26], а также проблема недоступных семантико-синтаксическому анализу используемых библиотечных функций и процедур обработки прерываний. Кроме этого, для многопоточного параллелизма, в задачах управления обычно реализуемого в виде кооперативной многопоточности, отсутствуют механизмы балансировки вычислительной нагрузки [27].

Таким образом, имеющиеся на рынке средства разработки ПО (в том числе ПО систем управления) не предоставляют возможностей априорно исследовать динамические характеристики создаваемых пользователем алгоритмов. А для систем, использующих многопоточный логический параллелизм, не решена проблема балансировки вычислительной нагрузки.

Язык процесс-ориентированного программирования Рефлекс

Язык Рефлекс, известный также под именем «Си с процессами» [28], ориентирован на программирование управляющих алгоритмов в промышленной автоматизации и робототехнике: для систем, предполагающих активное взаимодействие с внешней средой, технологическим оборудованием, физическими процессами через датчики и органы управления.

Базовые цели, которые ставились при разработке языка, – адекватность задачам управления, легкое изучение пользователем, комфортные программирование и сопровождение уже созданных программ.

Язык Рефлекс был выполнен как диалект языка Си, что обеспечивает простоту его изучения большинством практикующих программистов. Язык имеет англоязычный и русскоязычный синтаксис, а также допускает идентификаторы на русском языке, и это делает его привлекательным для отечественных пользователей.

В отличие от языка Си, где программы строятся как иерархия функций, базовое понятие языка Рефлекс – процесс. С математической точки зрения процесс представляет собой радикально модернизированный конечный автомат [29]. Процесс задается набором альтернативных функций (в программистском смысле), или, другими словами, процесс – это полиморфная функция. В отличие от полиморфизма, используемого в объектно-ориентированном программировании, полиморфизм процесса. *событийно-управляемый.* В отдельно взятый момент исполнения программы процесс представлен лишь одной из своих функций – *текущей функцией*. Текущая функция может переключаться по событиям. В качестве событий, приводящих к переключению текущей функции, могут выступать: тайм-ауты, изменения переменных и изменения текущих функций других процессов. Текущая функция задает, как следует реагировать на события (изменения во входных данных). Возможными реакциями может быть не только изменение выходных данных (управляющие сигналы для объекта управления), но и управление другими процессами – их запуск и остановка. Среди функций, задаваемых при описании процесса, выделяется *начальная функция* – функция, которая становится текущей при запуске процесса. В дополнение к функциям, задаваемым пользователем, в наборе функций процесса есть две *пассивные функции*: функции СТОП и ОШИБКА, в которых процесс не реагирует ни на какие события. Останов процесса означает смену его текущей функции на одну из пассивных. Назначение пассивных функций – организация взаимодействия между процессами через унифицированный механизм их запуска и средства контроля их завершения: успешного или неудачного (например, при обнаружении отказа оборудования процесс может просигнализировать о неудаче, установив в качестве своей текущей функции пассивную функцию ОШИБКА, что может быть проконтролировано запустившим его процессом).

Механизм пассивных функций – гибкое и простое средство для структуризации параллельно исполняемых процессов. Механизм тайм-аутов (который реализован через индивидуальный для каждого процесса счетчик времени, автоматически обнуляемый при смене текущей функции) позволяет генерировать временны́е события и тем самым обеспечивает возможность синхронизации алгоритма управления с физическими процессами на управляемом объекте.

Для комфортного программирования систем промышленной автоматизации в языке предусмотрены средства описания связей с датчиками и управляющими органами. Практическое использование языка Рефлекс в серии проектов по созданию управляющих комплексов выращивания монокристаллического кремния и корунда [30] показало его применимость для широкого класса объектов управления, а широкий набор специфических стратегий и приемов, не свойственных известным методам программирования, привел к необходимости выделить программирование на основе процессов в отдельный класс, получивший название *процесс-ориентированное программирование* (ПОП).

Основная реализация транслятора языка Рефлекс предполагает многопоточный логический параллелизм исполнения процессов в стиле *round-robin* [27] (рис. 1): исполнение алгоритма происходит циклически, на каждом цикле происходит последовательная активизация процессов. Активизация процесса приводит к исполнению их текущей функции, в которой определяются реакция на внешнее воздействие и, в частности, текущая функция процесса

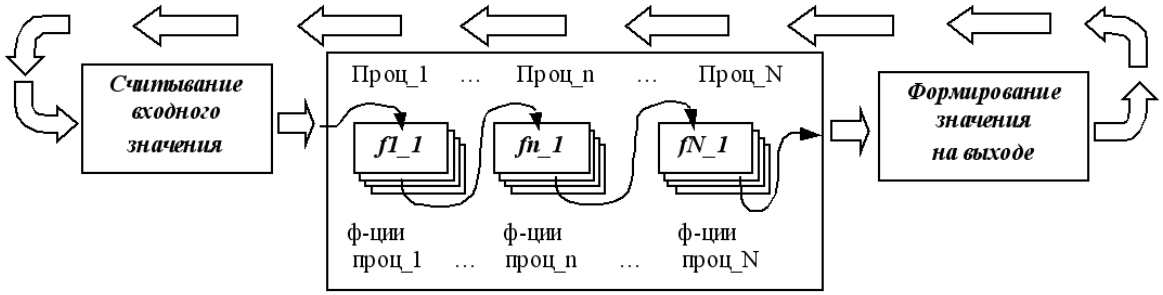

Рис. 1. Многопоточная реализация исполнения процессов

для следующего цикла. Периодичность активизации процессов должна быть в два раза выше, чем требуемое время реакции системы на наиболее критическое (в смысле скорости обработки) событие [14].

Многопоточная организация параллелизма процессов, не только облегчает создание системно-независимых (*stand alone*) приложений и их использование в широком спектре задач, в том числе во встраиваемых (*embedded*) системах, но и минимизирует накладные расходы на исполнение.

Разные динамические характеристики исполнительных органов, например, для реле время срабатывания исчисляется миллисекундами, а для клапанов – сотнями миллисекунд или десятками секунд, обусловливают и различные требования к временам реакции системы на события. А это означает не что иное, как отличия в допустимой периодичности активизации процессов: процессы, обслуживающие «медленные» устройства, могут активизироваться реже процессов, обслуживающих «быстрые» устройства.

Снижение периодичности активизации части процессов снижает ресурсоемкость алгоритма, но лишь потенциально, поскольку для этого должен наличествовать механизм настройки частоты активизации процесса – возможность балансировки вычислительной нагрузки алгоритма, реализованного принятым способом.

Варианты расширение синтаксиса языка Рефлекс

Проблема балансировки вычислительной нагрузки может быть решена за счет введения механизма, позволяющего задавать для процесса индивидуальный делитель базовой частоты активизации. Такая возможность может быть реализована через использование штатного операнда ТАЙМАУТ, что может обеспечивать не только единовременное задание частоты активизации отдельного процесса, но и изменение этой частоты во время функционирования, что активно используется, например, при настройке времени квантования цифровых ПИД-регуляторов. Однако, во-первых, это предполагает задание у процесса дополнительных функций и, следовательно, требует от программиста дополнительных усилий. Во-вторых, само по себе снижение частоты активизации процессов не приводит к снижению ресурсоемкости алгоритма. Например, если мы снижаем частоту активизации группы процессов в десять раз, то на первых девяти циклах ресурсоемкость алгоритма действительно радикально снижается, но на десятом цикле все процессы активизируются, что означает сохранение пиковой нагрузки системы в целом. Проблема может быть решена, если ввести индивидуальное смещение для каждого из рассматриваемой группы процессов. В этом случае часть процессов будет запускаться на первом цикле, часть на втором и т. д.

Введенные свойства процессов отражаются в математической модели гиперпроцесса [31] следующим образом: при задании процесса дополнительно указываются индивидуальный делитель частоты, индивидуальное смещение и текущее смещение, позволяющее обсуждать поведение процесса в динамике:

$$p_i = (F_i, f_i^1, f_i^{\text{cur}}, T_i^{\text{p}}, D_i, O_i, O_i^{\text{cur}}),$$

где F_i – множество функций-состояний *i*-го процесса; f_i^1 – начальная функция-состояние, $f_i^1 \in F_i$; f_i^{cur} – текущая функция-состояние, $f_i^{\text{cur}} \in F_i$; T_i^{p} – время отсутствия смены значения текущей функции-состояния процесса; D_i – индивидуальный делитель частоты процесса, $D_i \in \aleph$; O_i – индивидуальное смещение, $O_i \in \{0, 1, \ldots, D_i - 1\}$; $O_i^{\text{cur}} \in \{0, 1, \ldots, D_i - 1\}$ – текущее смещение, как и f_i^{cur} позволяющее рассуждать о процессе в динамике.

Индивидуальный делитель частоты устанавливает для *i*-го процесса длительность цикла, равную произведению $D_i * T_H$ (T_H – период активизации гиперпроцесса), что эквивалентно соответствующему увеличению времени реакции процесса на событие. При этом сохраняется преемственность с первоначальным определением гиперпроцесса, который является гиперпроцессом, состоящим только из процессов, имеющих индивидуальный делитель частоты, равный единице. Индивидуальное смещение определяет, в какой из тактов будет активизирован процесс. Активизация процесса происходит тогда и только тогда, когда остаток от деления числа прошедших с момента запуска гиперпроцесса тактов на D_i равен O_i. Процесс с расширенными свойствами обеспечивает возможность реализации механизмов автоматической балансировки вычислительной нагрузки.

На языковом уровне индивидуальный делитель тактовой частоты может задаваться опционально, посредством резервированного 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 гиперпроцесса. Присвоение индивидуальных смещений может производиться автоматически.

Балансировка вычислительной нагрузки

Для детерминированных систем, где механизмы своппинга, кэш-памяти и конвейеризации не вносят динамической неопределенности во время исполнения, формальное описание динамических характеристик систем, справедливое для модели гиперпроцесса получено в [14]. Общая методика вычислений подобна описанной в [24]. Максимальное гарантированное время реакции системы на внешнее событие ($T_{\text{макс}}$) определяется временем $T_{\text{накл}}$, затраченным на накладные расходы по организации функционирования алгоритма (связь с физическими устройствами ввода / вывода, организация параллельного исполнения), и временем, затраченным собственно на выполнение алгоритма ($T_{\text{алг}}$):

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

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);

$T_{\text{накл}}$ определяется как сумма времени, затраченного на организацию параллелизма, и времени, затраченного на обслуживание ввода / вывода.

Обслуживание ввода / вывода включает:

- 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 [14].

Если значение $T_{\text{накл}}$ не вызывает проблем с вычислением, то с введением индивидуальных делителей частоты процедура вычисления $T_{\text{алг}}$ усложняется.

Однако по-прежнему остаются два ключевых принципа:

- оценка ведется по пиковой нагрузке гиперпроцесса;
- при расчетах процесс характеризуется временем исполнения своей наиболее ресурсопотребляющей (пиковой) функции-состояния.

Поиск пиковой загрузки системы включает следующие этапы:

- определение пиковых функций-состояний процесса;
- статическую балансировку вычислительной нагрузки;
- определение максимальных времен исполнения интерферирующих элементов и нахождение искомого $T_{\text{алг}}$ как суммы найденных времен.

Определение пиковых функций-состояний процесса. Нахождение времени исполнения функции-состояния процесса заключается в определении наиболее ресурсопотребляющего пути исполнения алгоритма и подсчете времени исполнения инструкций этого пути [21; 22]. С введением операторов многовариантного разбора поиск такого пути усложнен появлением древовидной структуры пути. Однако в силу конструктивной невозможности организации внутри функции-состояния обратного течения управления древовидная структура возможных путей линеаризуется и может быть проанализирована во время одного из штатных проходов транслятора языка.

Пиковая функция-состояние процесса – функция-состояние с максимальным временем исполнения. Характеристическое время процесса, необходимое для дальнейших вычислений, равно времени исполнения пиковой функции-состояния. Характеристическое время *i*-го процесса обозначается $\mathrm{Max}(p_i)$.

Трудности, возникающие при подсчете этого времени:

- необходимость различать времена исполнения так называемых «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.

Статическая балансировка вычислительной нагрузки и расчет пиковой нагрузки процесса. Статическая балансировка осуществляется автоматически путем присвоения индивидуального смещения каждому из процессов, имеющих делитель смещения, отличный от единицы. Присвоение индивидуальных смещений производится на основании информации о временах исполнения пиковых функций-состояний по следующему алгоритму, названному субоптимальным алгоритмом:

- по описанию процессов определяется характеристическое время исполнения каждого процесса $\mathrm{Max}(p_i)$;
- процессы объединяются в группы по признаку равенства делителя частоты и в дальнейшем рассматриваются по группам независимо;
- внутри группы процессы ранжируются по характеристическим временам в убывающем порядке;
- для каждого из индивидуальных смещений, возможных для рассматриваемого делителя D, заводится счетчик суммарного времени T_i^{sum}, $i \in \{0, \ldots, D-1\}$;
- распределение индивидуальных смещений производится по процессам по порядку начиная с процесса с наибольшим характеристическим временем. Процессу p_j присваивается индивидуальное смещение, равное номеру k счетчика суммарного времени с минимальным значением ($O_j = k : T_k^{\text{sum}} = \mathrm{Min}(\{T_i^{\text{sum}}\})$), после чего счетчик суммарного времени увеличивается на величину $\mathrm{Max}(p_j)$.

Относительно предложенного подхода и алгоритма следует сделать ряд замечаний.

Замечание 1. Предложенный субоптимальный алгоритм обеспечивает нахождение истинного решения только для случая $N_D \le D + 2$, где N_D – число процессов с делителем D.

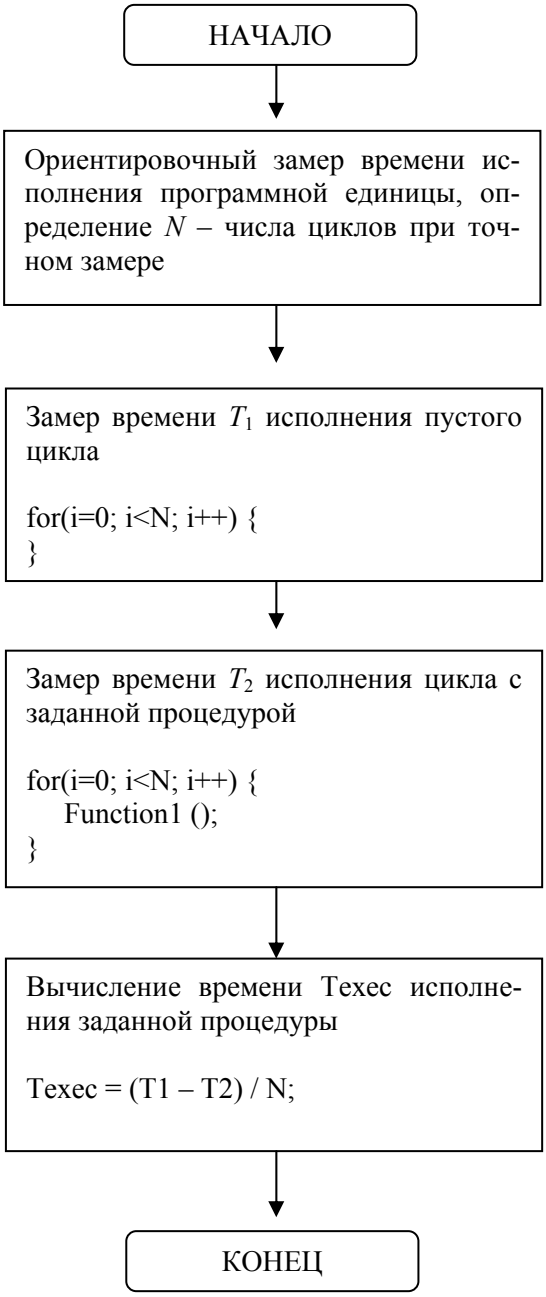

Рис. 2. Блок-схема вычисления времени исполнения исследуемой процедуры

Замечание 2. В общем случае решение может быть найдено только полным перебором вариантов, однако субоптимальный алгоритм предлагает разумный компромисс между простотой и качеством, которое достаточно в подавляющем большинстве случаев.

Замечание 3. Качество решения, получаемого субоптимальным алгоритмом, растет с ростом числа обрабатываемых процессов.

Замечание 4. Ресурсоемкость алгоритма может быть снижена за счет повышения частоты вызова процессов. Справедливость этого, на первый взгляд парадоксального, замечания может быть продемонстрировано на следующем примере. Если имеется непустое множество процессов с делителем 2 и один процесс с делителем 3, то установка для этого процесса делителя два приводит к снижению ресурсоемкости алгоритма за счет исключения эффекта интерференции.

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

Ввиду того, что $T_{\text{алг}}$ определено для случая выполнения процессами своих наиболее ресурсопотребляющих функций-состояний, реально наблюдаемое время реакции системы будет равно

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

Общий алгоритм определения временных характеристик для базовых языковых конструкций, служебных подпрограмм и библиотечных функций показан на рис. 2.

Особенность предлагаемого алгоритма – возможность снижения погрешности измерения за счет увеличения числа циклов *N*.

Поскольку автоматическое определение времени исполнения библиотечных процедур невозможно, генерируются тексты-шаблоны, в которых пользователь должен установить значения параметров и аргументов, 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, с.

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

Заключение

В работе была исследована возможность расширения языка процесс-ориентированного программирования Рефлекс 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. Предложен механизм индивидуального делителя частоты, позволяющий независимо регулировать время реакции на отдельное внешнее событие, и субоптимальный алгоритм балансировки вычислительной нагрузки. Описанные механизмы балансировки и статического анализа динамических характеристик алгоритма могут быть использованы при создании специализированных лингвистических средств и интегрированных сред разработки ПО систем управления в процесс-ориентированном стиле.

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. *Хоар Ч.* Взаимодействующие последовательные процессы: Пер. с англ. М.: Мир, 1989. 264 с.

4. *Harel D.* Statecharts: A Visual Formalism for Complex Systems // Science of Computer Programming. 1987. Vol. 8. No. 3. P. 231–274.

5. *Berry G.* The Foundations of Esterel // Proof, Language and Interaction: Essays in Honour of Robin Milner / Eds. G. Plotkin, C. Stirling, M. Tofte. MIT Press, 2000. P. 425–454.

6. *Milner R.* Communication and Concurrency // Series in Computer Science. Prentice Hall, 1989.

7. *Kaynar D. K., Lynch N., Segala R., Vaandrager F.* Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems // Proc. 24th IEEE International Real-Time Systems Symposium (RTSS'03). IEEE Computer Society Cancun. Mexico, 2003. P. 166–177.

8. *Kof L., Schätz B.* Combining Aspects of Reactive Systems // Proc. of Andrei Ershov Fifth Int. Conf. Perspectives of System Informatics. Novosibirsk, 2003. P. 239–243.

9. *Nixon P., Croll P.* The Functional Specification of occam Programs for Time Critical Applications // In Transputer and Occam Research: New Directions / Ed. by J/ Kerridge. IOS Press, 1993. P. 131–144.

10. *Ferdinand C., Heckmann R., Le Sergent T., Lopes D., Martin B., Fornari X., Martin F.* Combining a High-Level Design Tool for Safety-Critical Systems with a Tool for WCET Analysis on Executables // 4th European Congress on Embedded and Real Time Software (ERTS), 2008.

11. *Ju L., Huynh B. K., Roychoudhury A., Chakraborty S.* Performance Debugging of Esterel Specifications // International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS), 2008.

12. *Lei Ju, Huynh B. K., Chakraborty S., Roychoudhury A.* Context-Sensitive Timing Analysis of Esterel Programs // Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE. San Francisco, 2009. P. 870–873.

13. *Wilhelm R., Engblom J., Ermedahl A., Holsti N., Thesing S., Whalley D., Bernat G., Ferdinand C., Heckmann R., Mitra T., Mueller F., Puaut I., Puschner P., Staschulat J., Stenstrom P.* The Worst-Case Execution-Time Problem-Overview of Methods and Survey of Tools // ACM Transactions on Embedded Computing Systems (TECS). 2008. Vol. 7. Issue 3.

14. *Зюбин В. Е., Петухов А. Д.* Распределение вычислительных ресурсов в средах с многопоточной реализацией гипераавтомата // III Междунар. конф. «Идентификация систем и задачи управления» SICPRO'04 (Москва, 28–30 января 2004 г.): Сб. науч. тр. М., 2004. С. 446–463.

15. *Falk H., Lokuciejewski P.* A Compiler Framework for the Reduction of Worst-Case Execution Times // Springer Real-Time Systems. 2010. Vol. 46. Issue 2. P. 251–298.

16. *Головкин Б. А.* Расчет характеристик и планирование параллельных вычислительных процессов. М.: Радио и связь, 1983. 273 с.

17. *Куцевалов А. В., Куцевалов Д. В.* Организация процесса временного анализа программ // Программирование. 1978. № 4. С. 44–47.

18. *Капырин В. А., Халецкий А. К.* Способ оценки надежности функционирования программ реального времени // Программирование. 1982. № 3. С. 73–79.

19. *Гордиенко А. В.* Тестирование при оценке динамической корректности программ АСУ // Программирование. 1982. № 6. С. 48–52.

20. *Bates M. P.* Programming 8-bit PIC Microcontrollers in C: with Interactive Hardware Simulation. Elsevir, 2008. 278 p.

21. *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.

22. *Данильченко Л. С., Людвиченко В. А.* О получении априорных оценок времени решения задач программами на Фортране // Программирование. 1988. № 5. С. 56–60.

23. *Борзов Ю. В.* Тестирование программ с использованием символического выполнения // Программирование. 1980. № 1. С. 51–59.

24. *Димитров А. А., Каган Б. М., Крейнин А. Я.* Аналитические модели вычислительных систем реального времени с фоновыми задачами // Программирование. 1977. № 6. С. 60–65.

25. *Кринницкий Н. А., Чернова Т. Ф.* Об имитационном моделировании операционных систем // Программирование. 1981. № 3. С. 77–85.

26. *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.

27. *Зюбин В. Е.* Программирование информационно-управляющих систем на основе конечных автоматов: Учеб.-метод. пособие. Новосибирск, 2006. 96 с.

28. *Зюбин В. Е.* «Си с процессами»: язык программирования логических контроллеров // Мехатроника, автоматизация, управление. 2006. № 12. С. 31–35.

29. *Зюбин В. Е.* Процесс-ориентированный подход к программированию управляющих алгоритмов в среде LabVIEW // Промышленные АСУ и контроллеры. 2011. № 1. С. 39–45.

30. *Зюбин В. Е., Котов В. Н., Котов Н. В. и др.* Базовый модуль, управляющий установкой для выращивания монокристаллов кремния // Датчики и системы. 2004. № 12. С. 17–22.

31. *Зюбин В. Е.* Язык Рефлекс. Математическая модель алгоритмов управления // Датчики и системы. 2006. № 5. С. 24–30.

Материал поступил в редколлегию 29.07.2011

V. E. Zyubin

STATIC LOAD BALANCING IN PROCESS-ORIENTED PROGRAMMING FOR MULTITHREAD IMPLEMENTATION

The paper discusses the problem of load balancing in computing environment with multithread implementation of the process-oriented programming model. The paper presents seamless extension for the Reflex language. The paper proposes a technique for *a-priory* calculation of external event response time and a suboptimal method for static load balancing.

Keywords: process-oriented programming, control algorithms, load balancing for multithread implementation, tuning of the external event response time, a-priory estimation of dynamic behavior.