Lecture Notes in Computer Science 1985

Jack Davidson Sang Lyul Min (Eds.)

Languages, Compilers, and Tools for Embedded Systems

ACM SIGPLAN Workshop LCTES 2000
Vancouver, Canada, June 2000
Proceedings

Springer

Lecture Notes in Computer Science 1985

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Jack Davidson Sang Lyul Min (Eds.)

Languages, Compilers, and Tools for Embedded Systems

ACM SIGPLAN Workshop LCTES 2000
Vancouver, Canada, June 18, 2000
Proceedings

Springer

Preface

This volume contains the proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2000), held June 18, 2000, in Vancouver, Canada. Embedded systems have developed considerably in the past decade and we expect this technology to become even more important in computer science and engineering in the new millennium.

Interest in the workshop has been confirmed by the submission of papers from all over the world. There were 43 submissions representing more than 14 countries. Each submitted paper was reviewed by at least three members of the program committee. The expert opinions of many outside reviewers were invaluable in making the selections and ensuring the high quality of the program, for which, we express our sincere gratitude. The final program features one invited talk, twelve presentations, and five poster presentations, which reflect recent advances in formal systems, compilers, tools, and hardware for embedded systems.

We owe a great deal of thanks to the authors, reviewers, and the members of the program committee for making the workshop a success. Special thanks to Jim Larus, the General Chair of PLDI 2000 and Julie Goetz of ACM for all their help and support. Thanks should also be given to Sung-Soo Lim at Seoul National University for his help in coordinating the paper submission and review process.

We also thank Professor Gaetano Borriello of the University of Washington for his invited talk on Chinook, a hardware-software co-synthesis CAD tool for embedded systems.

January 2001

Jack Davidson
Sang Lyul Min

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Jack Davidson
University of Virginia, School of Engineering and Applied Science
Department of Computer Science
Charlottesville, VA 22903-2442, USA
E-mail: davidson@cs.virginia.edu

Sang Lyul Min
Seoul National University, Department of Computer Engineering
Seoul 151-742, Korea
E-mail: symin@dandelion.snu.ac.kr

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Languages, compilers, and tools for embedded systems : proceedings /
ACM SIGPLAN Workshop LCTES 2000, Vancouver, Canada, June 18, 2000.
Jack Davidson ; Sang Lyul Min (ed.). - Berlin ; Heidelberg ; New York ;
Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo :
Springer, 2001
(Lecture notes in computer science ; Vol. 1985)
ISBN 3-540-41781-8

CR Subject Classification (1998): D.4.7, C.3, D.3, D.2, F.3

ISSN 0302-9743
ISBN 3-540-41781-8 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author
Printed on acid-free paper SPIN: 10781420 06/3142 5 4 3 2 1 0

Preface

This volume contains the proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2000), held June 18, 2000, in Vancouver, Canada. Embedded systems have developed considerably in the past decade and we expect this technology to become even more important in computer science and engineering in the new millennium.

Interest in the workshop has been confirmed by the submission of papers from all over the world. There were 43 submissions representing more than 14 countries. Each submitted paper was reviewed by at least three members of the program committee. The expert opinions of many outside reviewers were invaluable in making the selections and ensuring the high quality of the program, for which, we express our sincere gratitude. The final program features one invited talk, twelve presentations, and five poster presentations, which reflect recent advances in formal systems, compilers, tools, and hardware for embedded systems.

We owe a great deal of thanks to the authors, reviewers, and the members of the program committee for making the workshop a success. Special thanks to Jim Larus, the General Chair of PLDI 2000 and Julie Goetz of ACM for all their help and support. Thanks should also be given to Sung-Soo Lim at Seoul National University for his help in coordinating the paper submission and review process.

We also thank Professor Gaetano Borriello of the University of Washington for his invited talk on Chinook, a hardware-software co-synthesis CAD tool for embedded systems.

January 2001

Jack Davidson
Sang Lyul Min

Program Committee

Neil C. Audsley	University of York, UK
Jack Davidson	University of Virginia, USA
Alan Davis	Texas Instruments, USA
Susanne Graf	Verimag, France
Seongsoo Hong	Seoul National University, Korea
Alan Hu	University of British Columbia, Canada
Inhye Kang	Soongsil University, Korea
Tei-Wei Kuo	National Chung Cheng University, ROC
Jane Liu	University of Illinois, USA
Sharad Malik	Princeton University, USA
Peter Marwedel	University of Dortmund, Germany
Sang Lyul Min	Seoul National University, Korea
Ramesh Peri	Intel, USA
Peter Puschner	Technical University of Vienna, Austria
Gang-Ryung Uh	Lucent Technologies, USA
Wang Yi	Uppsala University, Sweden

List of Reviewers

Tobias Amnell
Pavel Atanassov
Neil C. Audsley
Iain Bate
Dean Batten
Guenter Bauer
Bryan Bayerdorffer
Johan Bengtsson
Guillem Bernat
Clark Coleman
Jack Davidson
Alan Davis
Sri Doddapaneni
Julien d'Orso
Jakob Engblom
Heiko Falk
Jose Fridman
Gregor Goessler
Susanne Graf
Jason Hiser
Marc Hoffman
Seongsoo Hong
Alan Hu
Sanjay Jinturkar
Bengt Jonsson
Inhye Kang
Changhwan Kim
Chanho Kim
Saehwa Kim
Tei-Wei Kuo
Yassine Lakhnech
Fredrik Larsson
Chang-Gun Lee
Seung-Hyoun Lee
Sheayun Lee
Rainer Leupers
Wei Li
Sung-Soo Lim
Jane Liu
Sharad Malik
Peter Marwedel
Chris Milner
Sang Lyul Min
Sven Olof Nystrom
Roman Pallierer
Jungkeun Park
Ramesh Peri
Peter Puschner
Kevin Scott
Yangmin Seo
Joseph Sifakis
Christopher Temple
Gang-Ryung Uh
Lars Wehmeyer
Wang Yi

Table of Contents

Hardware

Work in Progress

Randomization-Based Approaches for Dynamic Priority Scheduling of Aperiodic Messages on a CAN Network

Lucia Lo Bello and Orazio Mirabella

Department of Computer Science and Telecommunications
University of Catania
V.le A. Doria 6, I-95125 Catania, Italy
{llobello, omirabel}@iit.unict.it

Abstract. The Controller Area Network is a well-known communication bus widely used both in automotive applications and in other environments, as distributed embedded systems, to support the exchange of control messages. Any contention on a CAN bus is deterministically resolved on the basis of the priority, which is fixed and encoded in the identifier field of the frame. Fixed priorities are not the best solution for event-driven applications, such as many distributed embedded control systems, which feature significant reactive behaviour and produce aperiodic traffic which can have real-time constraints. Depending on the priority some aperiodic objects may experience unusually higher access delays compared with those from other objects. This problem can be overcome by dynamically assigning identifiers to aperiodic messages. This paper proposes the use of randomization to provide dynamic priority assignment in a CAN network. The aim of this work is to investigate how the management of aperiodic traffic in a CAN system can be improved using randomization. Two different strategies for aperiodic traffic scheduling on a CAN network are presented and evaluated by simulation.

1 Introduction

The Controller Area Network (CAN) [1][2] communication bus was originally designed for use within road vehicles to solve cabling problems arising from the growing use of microprocessor-based components in vehicles. CAN is widely used in automotive applications and, as it is highly versatile, it can also be adopted in other environments. Thanks to the low cost of CAN bus adaptors and its ability to support real-time communication, CAN can be used as an embedded control networks (ECN) to connect several Function Control Units in a distributed embedded system.

J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 1-18, 2001.

For example, in an intelligent transportation system for real-time monitoring of road traffic conditions, CAN can be used in a passenger vehicle equipped with microprocessors and connected to the internet via a wireless network.

Several works have dealt with high-level protocols for the CAN bus [3][4][5][6] and a recent work [7] has proposed an environment specific CORBA for CAN-based distributed embedded systems.

One of the reasons of the success of the CAN lies in the non destructive priority based bus arbitration mechanism it implements. Any message contention on a CAN bus is deterministically resolved on the basis of the priority of the objects exchanged, which is encoded in the identifier field of the frame. Thus the priority of a messages is the priority of the object it contains, expressed by the identifier. The short length of the bus allows all nodes to sense the same bit, so that the system can behave as a kind of large AND-gate, with each node able to check the output of the gate. The identifier with the lowest numerical value has the highest priority, and a non-destructive bitwise arbitration provides collision resolution.

The priority of a CAN message is static, system-wide unique and it is assigned during the initial phase of system design. As the CAN application layer proposal [8] provides a direct mapping between the Data Link level object (COBs, Communication Objects) and the information at the Application level, each connection is given a different and fixed priority.

As it is known, fixed priorities are suitable to support time-driven interactions (as, for example, periodic exchanges between a master unit and a set of slave devices). Several strategies can be adopted to assign identifiers to periodic messages so that all the application deadlines can be met, and the results of a scheduling analysis developed in order to bound the worst case response time of a CAN message can be used. In [9][10] the authors addressed in detail the analysis of real-time communications in CAN, assuming fixed priorities for message streams. The Deadline Monotonic (DM) priority assignment [11] can be directly implemented in a CAN network, by setting the identifier field of each message stream to a unique priority, according to the DM rule.

While CAN is highly suitable to support periodic traffic, its fixed-priority-based approach is not able to support efficiently aperiodic exchanges, both real-time and non-real-time, typical of event-driven applications, such as many real-time embedded control systems, which feature significant reactive behaviour and produce aperiodic traffic which can have real-time constraints [12]. Due to the lack of timing information for the real-time aperiodic exchanges, it is not possible to find a static way to determine CAN identifiers for this kind of objects so that their deadlines can be always met. In addition, depending on the assignment of identifiers, there can be a repetition of undesirable sequences for aperiodic messages and it is possible that certain aperiodic objects may experience unusually higher access delays compared with those of other objects.

This problem can be overcome by dynamically assigning identifiers to aperiodic messages. In [13][14] the authors analyzed how the non pre-emptive Earliest Deadline First (EDF) [15] could be used to schedule CAN messages. They proposed a mixed traffic scheduler (MTS), which uses the standard 11-bit format for the identifier field (provided in CAN Version 2.0A). The key idea of the MTS scheduler is to assign identifiers to different messages so that they reflect the messages' deadlines. As CAN requires that each message must have a unique identifier field, they suggested the division of the identifier field into three sub-fields.

Another work [16] introduced dynamic priorities in CAN networks through a mechanism which makes transmissions of aperiodic messages occur in a round-robin way. However, round-robin may not always be the best solution to handle aperiodic traffic in a CAN network. For example, depending on the application, it might be needed that some objects be processed more quickly than other ones, or with a better 'quality of service' (in terms of throughput or bus access delay). A round-robin scheme cannot provide preferential service of aperiodic messages needed to deal with such kind of situation.

For this reason, this paper proposes new solutions to provide dynamic priority assignment in a CAN network, based on randomized choice of priorities for exchanged objects. An algorithm is said to be randomized if its behaviour depends not only on the input, but also on values produced by a random-number generator. Use of randomization in generating dynamic priority for an aperiodic object avoids deterministic behaviours and leads to a probability distribution over the admissible priority levels for that object, i.e. to a probability distribution of the access delay the object will experience. As it will be shown in the paper, the introduction of randomization makes it possible to provide, depending on the application requirements, either a preferential service for some class of objects or fairness. Two different approaches are proposed, called Randomized Adjacent Priority (RAP) and Randomized Priority with Overlapping (RPO). They are very simple to implement and differ in the strategy used to generate aperiodic messages' priorities.

2 Dynamic Priority Assignment in a CAN Network

As said in Sect.1, while CAN is highly suitable to support periodic traffic, its fixed-priority-based approach is not able to support efficiently aperiodic exchanges typical of event-driven systems. A point against fixed-priority assignment comes from evidence that, when the workload is close to the nominal bandwidth of the system, the transmission delays experienced by the objects vary significantly according to their priority (i.e. according to the identifiers assigned to them). Finally, there is no way of providing for fair bandwidth sharing between non real-time applications without adopting poor solutions (e.g. limiting in advance the bandwidth allocated to each object) which affect the overall efficiency. For this reason there is a need to find new solutions that, without altering the ability of the CAN protocol to support periodic object exchanges correctly, would be efficient for aperiodic traffic as well.

In [16] the introduction of dynamic priorities was proposed to better support sporadic or aperiodic object exchanges, even in the case of system overload. In fact, in such conditions, objects with lower identifiers (i.e. having a higher priority) can monopolize all the bandwidth thus heavily affecting the transmission of objects with higher identifiers (lower priority). The use of dynamic identifiers allowed the authors to introduce mechanisms that make the protocol more fair. They presented two approaches: the Distributed Priority Queue, which requires the management of a global distributed queue where all the nodes are virtually inserted, and the Priority Promotion (PP), based on priority increase. The latter is based on the idea of assigning each node a priority level, called station priority, which depends on the object to be transmitted and progressively increases proportionally to the time that has

elapsed since the node first attempted to transmit the current frame. In PP the extended identifier format (Version 2.0B) is used which, as foreseen by the CAN standard, is 20 bits longer than the normal format (Version 2.0A, which is shown in Fig.1a) and comprises a *base identifier* plus an identifier *extension* (Fig. 1b)[1].

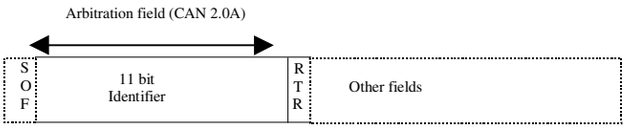


Fig. 1a. Arbitration field for the 2.0A CAN protocol

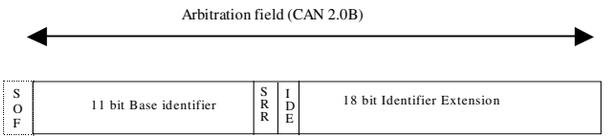


Fig. 1b. Arbitration field for the 2.0B (Extended) CAN protocol

As Fig. 1b shows, when an extended identifier field is used (V 2.0B), two sub-identifiers are available in the frame. The first is dynamic; it is used by the Medium Access Control (MAC) sub-layer to solve any contention (it is called a base identifier and is 11 bit long as the 2.0A CAN identifier), while the second is static and contains the effective identifier of the object being exchanged. It is used by the Logical Link Control (LLC) in the Data Link Layer to filter the content of a message. In fact, CAN messages produced by any node do not contain the address of the destination node. It is the consumer's responsibility to filter messages and select only those that are of interest to the node.

In [16] a service priority is defined for the PP protocol, representing the degree of urgency assigned to objects to be transmitted, which is specified by the application on each transmission service invocation. Service priority is encoded in a two-bit field called priority class (PC) (see Fig. 2), while the remaining bits of the base identifier encode the station (node) priority.

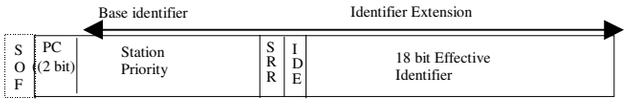


Fig. 2. Arbitration field for the Priority Promotion protocol

[1] In [16] it was proved that this little increment in the frame size introduces only a small additional cost.

In PP, a node priority is dynamically updated to indicate the precedence the node has in sending a frame when a collision occurs. Each node stores its priority level in a local variable which is dynamically reassigned by the MAC protocol, so that the priority is increased by 1 when a collision occurs and is set to the lowest value for the associated service priority class (each class has its own lowest value) when a transmission is successfully completed.

This protocol organizes traffic into at most four priority classes (time critical, high, low and time available) and messages inside each class are transmitted on the basis of a round-robin approach. In many situations, round-robin mechanism is not the best solution. An application process has to be given the possibility of transferring more data than the others at any one time (for example, it has to transfer a large table to calibrate a sensor, or download a program into a slave node). Again, depending on the application, it might be needed that some objects be given a better quality of service in terms of throughput and access delay.

The main drawback of the scheme proposed in [16] is that it can not provide preferential service of aperiodic messages needed to deal with such kind of situation. For this reason, this paper proposes new solutions to provide dynamic priority assignment in a CAN network, based on randomized choice of base identifiers for messages.

3 Reasons for Randomization

An algorithm is said to be randomized if its behaviour depends not only on the input, but also on values produced by a random-number generator. Randomization has the property that no particular input elicits its worst case behaviour, that is, no single input will always evoke worst case behaviour, and guarantees good-average case performance, no matter what keys are provided as input [17]. A suitable use of randomization therefore can provide more flexibility and effectiveness in dynamic priority assignment.

Use of randomization avoids deterministic behaviours, which in many cases may lead to transmission sequences that are unfavourable to some low priority objects. When randomization is used in generating the base identifier of an object, the induced outcome is itself random, leading to a probability distribution over the admissible priority levels for that object. This also leads to a probability distribution of the access delay that the exchanged object will experience.[2]

---

[2] It is possible that, due to randomization, different messages have the same base identifiers. For example, for two base identifiers generated independently from each other in the same range of length N according to a uniform probability distribution, the probability of obtaining the same value is equal to $1/N^2$. However, our approach does not require different objects to be assigned different priorities to work correctly. If two or more messages having the same base identifier start transmitting at the same time, the contention is resolved based on the effective identifier field. The same situation occurred in [16], both at system start-up, when all the nodes were given the same identifier value, and every time a new node joined the bus.

As random functions generate numbers within a given range (which can be specified by a generation range), with a uniform probability distribution, by using the theory of probability it is possible to obtain the distribution of the access delays of the various messages according to the randomized dynamic priorities assigned to them. As a result, a suitable method to determine the length of the generation range in order to provide the required performance level can be found. For aperiodic low-priority real-time traffic the performance target can be that of probabilistically guaranteeing an upper bound on the access delay for each object. For non real-time traffic, the target can be, for example, that of providing an acceptable average access delay for objects, by giving them a non-null probability of being transmitted even in the presence of unfavourable workload condition on the network. In our scenario randomization can be applied in different ways for diverse purposes.

Two different approaches are proposed in the paper, called Randomized Adjacent Priority (RAP) and Randomized Priority with Overlapping (RPO).

4 The Basic Idea

The two approaches proposed in this paper, RAP and RPO, have in common in that base identifiers (and consequently, priorities) are assigned to messages according to the requirements specified by the application for the exchanged objects. Moreover, both the algorithms use a random function to generate the object priority and encode it using all 11 bits of the base identifiers shown in Fig. 1b[3].

Each base identifier is an integer belonging to a range $[N_{min}, N_{max}]$ fixed by the CAN protocol. To support different priority service classes, we split the entire domain of identifiers into three main parts. The first part, indicated as $[N_{aperiodic\_hp}, N_{periodic} -1]$, which features the lowest identifiers (i.e. those giving maximum priority to messages), is reserved for high-priority real-time aperiodic messages (e.g. alarms). As there are normally very few of these messages, it is reasonable to assign them a limited range (e.g. from 0 to 100 or less). A second range of identifiers $[N_{periodic}, N_{aperiodic\_lp} -1]$ is reserved for periodic traffic. As periodic traffic is known a priori, we can assign a fixed priority to the messages, and choose the identifiers so that deadlines can be met using suitable approaches following the analysis made in [9][10]. Randomization is not needed for both these kinds of traffic.

Finally, we use a third range of identifiers $[N_{aperiodic\_lp}, N_{max}]$ for both low-priority aperiodic traffic real-time[4] and non real-time traffic. As will be seen in what follows, the range $[N_{aperiodic\_lp}, N_{max}]$ can be further divided into sub-ranges, in order to provide

[3] The CAN protocol states that 2.0B controllers are completely backward compatible with 2.0A controllers and can transmit and receive messages in either format. It is also possible to use both V 2.0A and V2.0B controllers on a single network, provided that V.2.0A controllers of the "2.0B passive type" are used.

[4] Here we assume that real-time low priority traffic has soft deadlines. Through a suitable choice of randomization, it can be given an upper bound for the access delay in a statistical way.

several different service priority classes for aperiodic low priority traffic. For this kind of traffic we can generate dynamic node priorities by means of random numbers following the RAP and RPO policies described above.

The difference between RAP and RPO lies in the strategy used to generate objects priority. The two algorithms are extremely simple to implement, as they do not require the management of distributed queues or priority increase strategies as in [16], but only the generation of random numbers within a pre-defined range.

5 Randomized Adjacent Priority Protocol

As said above, several different priority classes for low-priority aperiodic traffic can be obtained through suitable partitioning into sub-intervals of the range [$N_{aperiodic\_lp}$, N_{max}]. This partitioning can be done off-line according to the requirements of the application. In the RAP algorithm we assume that the various ranges are disjoint, thus that the service classes obtained are disjoint. When a node has to transmit an object with a service priority class specified by the application, it picks up an identifier (node priority) generated using randomization in the range associated with the given class.

For example, let us suppose that only two service classes have to be supported for aperiodic low priority traffic. Thus, let us refer to an identifier space comprising N_{max} - $N_{aperiodic\_lp}$ different values, divided into two intervals of length s_1 and s_2 respectively, associated to the first and the second service class. In the RAP algorithm identifiers for the first class (id_1) are generated using the formula

$$id_1 = N_{aper\_lp} + rand\ (s_1) \tag{1}$$

so $id_1 \in A$, where $A=[\ N_{aper\_lp},\ N_{aper\_lp} + s_1 - 1]$, while identifiers for the second class (id_2) are generated using the formula

$$id_2 = N_{aper\_lp} + s_1 + rand\ (s_2) \tag{2}$$

so $id_2 \in B$, where $B=[\ N_{aper\_lp} + s_1,\ N_{aper\_lp} + s_1 + s_2 - 1]$.

As it can be seen, $A \cap B = \varnothing$. It is immediate to extend the above presented formulas if more than two service classes are present.

As the probability is uniformly distributed over the generation range, thanks to randomized generation of objects' priorities, messages belonging to the same service priority class have the same probability of being successfully transmitted in the case of conflict on the bus. RAP can therefore provide fairness, but it is more flexible than the algorithms in [16]. However, it has the same limits, as it is not possible to give only a preferential service (that is, a greater probability of success in the case of conflict, but no deterministic success) to objects belonging to higher priority classes. That is, in the event of collision messages having a higher service priority will always win over messages having a lower service priority. For this reason the RPO mechanism has been introduced.

6 Randomized Priority with Overlapping

The scenario for the RPO protocol is the same as that described for RAP as regards the partitioning of the entire domain of base identifiers into three main parts. Again, RPO envisages that priority is assigned to a message dynamically, according to the service class (specified by the application), when it is generated. However, the RPO mechanism generates object priority using randomization over a variable-length range, being longer or shorter according to the requirements of the application. That is, if a node has to transmit a low priority aperiodic message M_1 with a preferential service in order to comply with the 'urgency' specified by the application, it will use the random function in such a way that it will return an identifier that is as low as possible in the range $[N_{aperiodic\_lp}, N_{max}]$.

Thus randomization in the generation of an identifier for M_1 will be performed so that the value returned for M_1 will be as close as possible to $N_{aperiodic\_lp}$, for example, in a range $[N_{aperiodic\_lp}, N_{aperiodic\_lp}+\delta]$, $\delta \in N$ (δ has to be small, as lower identifiers give higher priorities). If, on the other hand, a generating node has to transmit a message, M_2, for which the application did not specify particular requirements, it will pick up the identifier for M_2 using a random function such that it will return a value which may belong to a wider range $[N_{aperiodic\_lp}, N_{aperiodic\_lp}+D]$, with $D \in N : D > \delta$. The advantage of using randomization in this way is that there is a non-null probability that message M_2 will have a priority higher than M_1.

Let us consider the same situation already described for RAP. We refer to an identifier space comprising $N_{max} - N_{aperiodic\_lp}$ different values, divided into two intervals of length s_1 and s_2 respectively, with $s_2 > s_1$, associated to the first and the second service class. In the RP0 algorithm identifiers for the first class (id_1) are generated using the formula

$$id_1 = N_{aper\_lp} + rand(s_1) \quad (3)$$

so $id_1 \in A$, where $A=[N_{aper\_lp}, N_{aper\_lp} + s_1 - 1]$, while identifiers for the second class (id_2) are generated using the formula

$$id_2 = N_{aper\_lp} + rand(s_2) \quad (4)$$

so $id_2 \in B$, where $B=[N_{aper\_lp}, N_{aper\_lp} + s_2 - 1]$.

In this case, $A \cap B \neq \emptyset$, as $A \subset B$. It is immediate to extend the above presented formulas if more than two service classes are present.

Differently from RAP and the approaches in [16], RPO allows a "controlled" degree of unfairness between aperiodic objects in the range $[N_{aperiodic\_lp}, N_{max}]$: this means that if different service classes are provided in that range, lowest priority class objects do have a chance of winning a contention and then of being transmitted, even in the presence of objects of other classes. We underline that RPO preserves high-priority exchanges (both aperiodic real-time and periodic) as they are reserved identifiers in other ranges (as said in Sect.5) which are not affected by randomization

and feature the lowest identifiers. RPO can control the fairness of the CAN protocol in managing transmission of all the aperiodic low-priority objects. In this way it is possible, by using different but partially overlapping ranges, to assign the various objects different transmission success probabilities.

In low workload conditions randomization does not affect the throughput of any object, because if the bandwidth is sufficient all the objects will be transmitted. However, when the offered traffic approaches the saturation point, objects whose identifiers are generated in smaller ranges have a higher probability of being successfully transmitted, and thus of having a higher throughput, but they do not totally prevent the transmission of the other objects.

7 Remarks on the Identifier Ranges

As we stated earlier, 11 bits are devoted to generating random identifiers and they represent the first part of the arbitration field of the frame in Fig. 1b, i.e. the one used in our approach to solve conflicts caused by collisions. The other 18 bits, which make up the data item's effective identifier, are used to solve the conflict only if it is not solved by the first 11 bits (i.e. when randomization returns the same value). If we generate two random numbers, "a" and "b" with a<b, in the event of a collision the conflict will be solved by the 11 bit representing a and b, with no need to use the remaining 18 bits. If, on the other hand, a=b, then the MAC layer will continue the arbitration using the remaining 18 bits; as these vary according to the objects (effective identifiers are unique throughout the network) a winner will be chosen. In this case, however, the success will not be random, because it depends on the value of the effective identifier which was statically assigned during configuration of the system.

In both approaches, each time a contention is resolved, priorities of losing messages are re-assigned. To exploit to the best our randomization-based approaches the generation of random numbers should give different values each time.

Since the identifier space used for the purpose is limited to 11 bits, and as it can be divided into several sub-intervals, it is of use to consider the effect that this subdivision may have on the generation of the random numbers.

Fig. 3 shows the delay times the objects of a single class undergo according to their effective identifier in normal workload conditions (i.e. unsaturated network). The various curves refer to different ranges used for the generation of random identifiers for that class. In the figure, R is the length of the range used for random numbers generation, which represents the number of distinct identifiers that an object belonging to that class can be assigned.

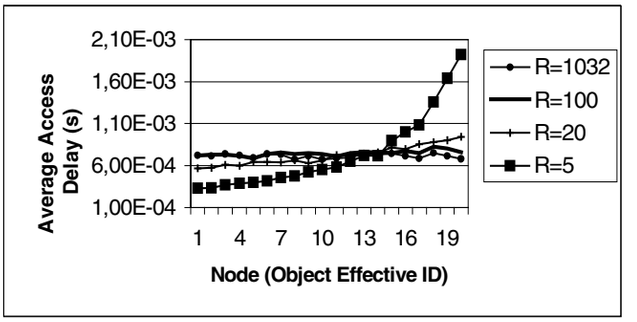


Fig. 3. Average access delay versus number of objects in a class

As can be seen, as long as the identifier space is high there are no significant differences between the various curves.

When randomization is performed on smaller ranges (i.e. R=5), however, the probability of generating the same numbers increases and thus any conflict will be solved on the basis of the effective identifiers alone. In this case, the delay increases along with the value of the effective identifier.

8 Performance Evaluation

To evaluate the performance of the RAP and RPO protocols we developed an ad hoc simulator, defining several application scenarios that would throw light on the differences with respect to the traditional CAN protocol.

In the simulation here presented, we envisaged a system with 20 nodes, which generated aperiodic traffic according to an exponential generation with the same mean value of the interarrival time of the frames for all the nodes. For simulation purposes, as in [16], it was also assumed that each node could send only one object, whose effective identifier was taken equal to the node number plus an offset. This offset was dependent on the length of the range reserved for identifiers used for aperiodic high priority and periodic traffic. Thus messages sent by a node n_i have a greater priority than messages transmitted by a node n_j, if i is less than j. In our trials we set workload values both below saturation, with the offered traffic below the nominal bandwidth of the system, and slightly over the bandwidth saturation point, with the offered traffic exceeding the nominal bandwidth of the system of a 10 percent. Our aim was to highlight the difference between RAP and RPO in throughput and average access delay. The access delay is defined as the time elapsing between the transmission request to the successful transmission of the first bit of the frame over the network.

The priority values were distributed by assigning base identifiers to low priority aperiodic traffic in the range $[N_{aperiodic\_lp}, N_{max}]$. Below we will discuss the results obtained with reference to non real-time aperiodic traffic alone, as it is the traffic type on which randomization has the greatest impact.

8.1 Performance of the RAP Algorithm

As discussed previously, the aim of the RAP and RPO methods is to implement strategies based on dynamic priority assignment to handle low priority aperiodic traffic when it is necessary to serve objects with different requirements in terms of throughput and access delay. For simulation purposes we assumed that, according to the application requirements, the aperiodic low priority messages (i.e. the objects to be exchanged) could be split into two different classes. Thus the traffic was given two different priority levels by subdividing the range for randomized generation of base identifiers $[N_{aperiodic\_lp}, N_{max}]$ into two intervals of different size $[N_{aperiodic\_lp}, N_{aperiodic\_lp1}-1]$ and $[N_{aperiodic\_lp1}, N_{max}]$. The results presented here refer to an identifier space for aperiodic low priority traffic comprising about 1032 different values, divided into two intervals s_1 and s_2 of size s_1=100 and s_2=932 respectively, thus we assumed $N_{aperiodic\_lp1}$=1100.

For a better comprehension of the graphs, in all the figures on the x-axis we do not indicate the effective identifier of the exchanged object, but the number of the node generating it (as we assume that each node can send only one object). That is, we subtracted to the effective ID of each object the offset representing the range of identifiers reserved to the other kinds of traffic in the network. This choice does not affect the values obtained. Objects belonging to the higher priority class (indicated in the graphs with numbers from 1 to 10), which were assigned base identifiers randomly generated in a range from 1001 to 1100, obtained lower average delays than the other group of objects (11 to 20), which were assigned base identifiers in the range (1101-2032). Within a single generation range the delay is constant, while the difference in the delay values between the two distinct classes reflects the different requirements of the two classes. In normal workload conditions (i.e. below saturation) there is no difference in throughput between the two priority classes, as Fig. 5 shows, because there is enough bandwidth to transmit all the offered traffic.

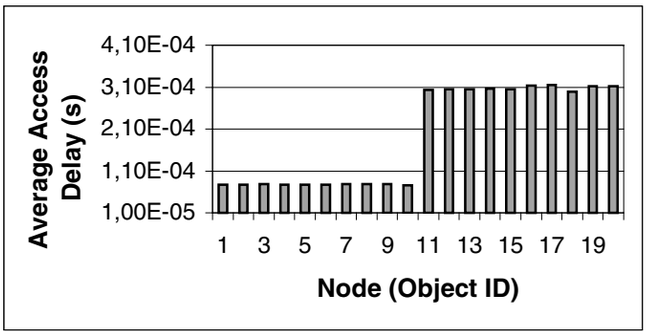


Fig. 4. Average access delays in normal workload conditions

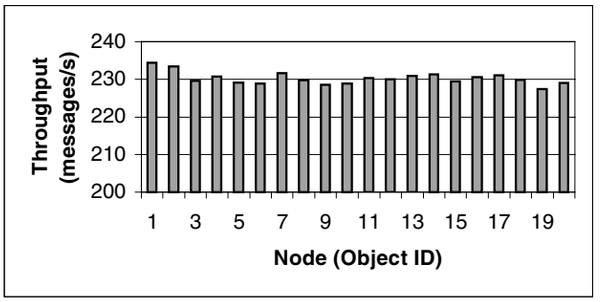


Fig. 5. Throughput in normal workload conditions

In heavy workload conditions, objects in the second class experience very long delays, as it is shown in Fig. 6, as the bandwidth is mainly monopolized by objects in the first class, as it is confirmed in Fig. 7, which shows the throughput obtained. As can be seen, when the system approaches saturation the traffic of first class objects takes bandwidth away from traffic of the other class. As a result, all the objects belonging to the first class are transmitted, whereas some second class objects can exploit only the remaining part of the bandwidth, which is not enough to serve all the offered workload.

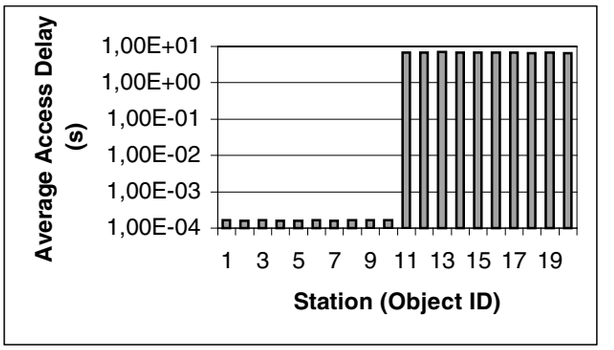


Fig. 6. Average access delays in saturation conditions

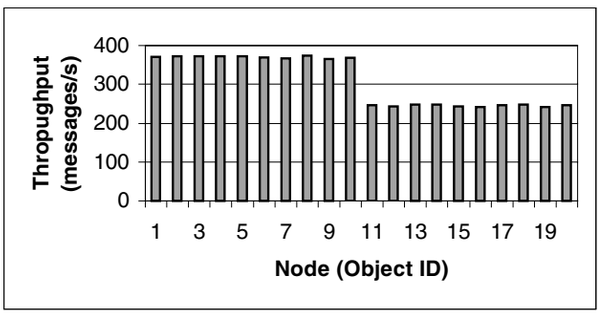


Fig. 7. Throughput in saturation conditions

This bandwidth monopolization is the main limit of using priorities to meet the different requirements of various kinds of aperiodic traffic. Priority is a rigid mechanism that does not take the requirements of lower-priority traffic into account.

As an effect of the different bandwidth exploitation for the two classes, the access delays are distributed in a very different way. This is confirmed by Figs. 8 and 9, which show the delay distribution, for first and second class objects respectively, obtained under a high workload, but below saturation. For the first class object (Fig. 8) the curve is more sharp and most of the delay values are less than 0.2 ms. On the contrary, the curve in Fig. 9 is bell-shaped and centered around the value of 0.22 s.

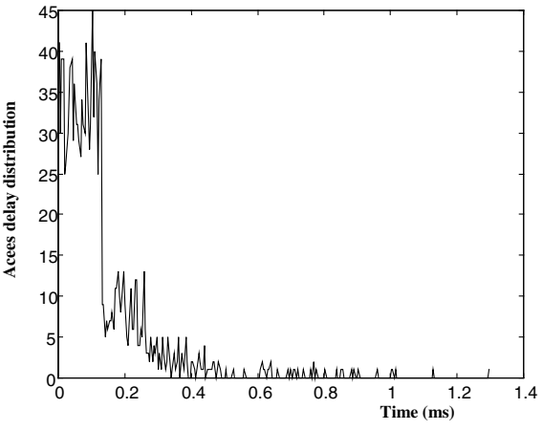


Fig. 8. Access delay distribution for a first class object

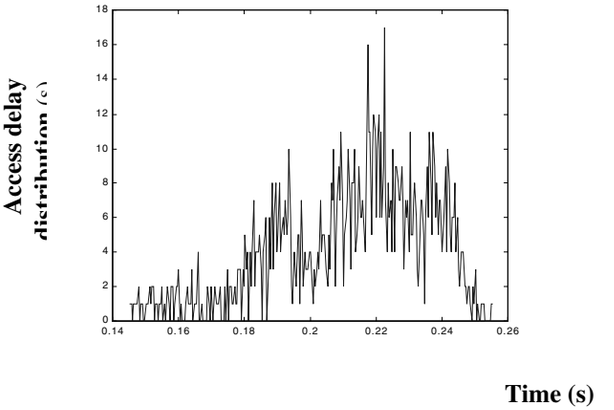


Fig. 9. Access delay distribution for a second class object

8.2 Performance of the RPO Algorithm

The RPO method does not organize the traffic by priority in a rigid way, but randomly generates identifiers in partially overlapping ranges.

The effect of this partial superposition is to smooth the priorities and reduce the differences in performance between the two different classes. Two classes were defined, corresponding to (base) identifiers from 1001 to 1100 (the first class) and from 1101 to 2032 (the ranges' size was 100 and 1032 respectively).

Two simulations were performed: in the first one, the offered traffic was below the nominal bandwidth of the system, while in the other one the bus was in saturation conditions.

In the first simulation, as Fig. 10 shows, the obtained throughput was the same for the two classes, whereas the average access delay was dependent on priority, as it is shown in Fig. 11.

In saturation conditions the throughput is different for the two classes, because the first class consumes more bandwidth than the other one. In simulation we used two ranges, (100, 1032) and (500, 1032) respectively. In Figs. 12 and 13 the histogram labeled with 'a' was obtained using the first range, while histogram 'b' refers to the second range.

Fig. 12 shows that there is a significant difference between the different priority classes for what the average access delay is concerned. In particular, as histograms in Fig. 12 show, the use of two ranges of very different length increases the difference in access delay values obtained for objects belonging to different classes.

On the contrary, if the gap between the two ranges is low, as in case 'b', the access delays obtained for the two classes are closer. This effect is less evident for the throughput, as it is shown in Fig. 13, where first class objects achieved most of the bandwidth in both cases 'a' and 'b'. There is not a significant difference between cases 'a' and 'b' because the advantage the first class is given (in terms of a greater probability of success) is sufficient to transmit all the objects belonging to this class.

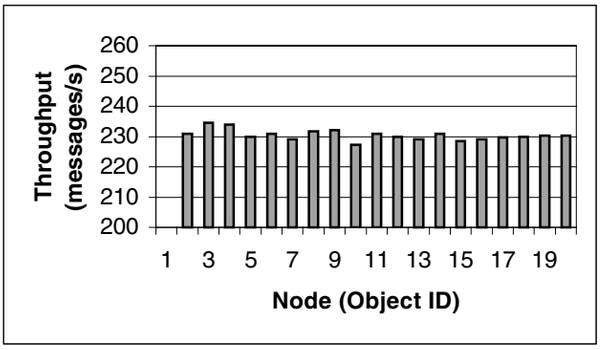


Fig. 10. RPO - Throughput in normal workload conditions

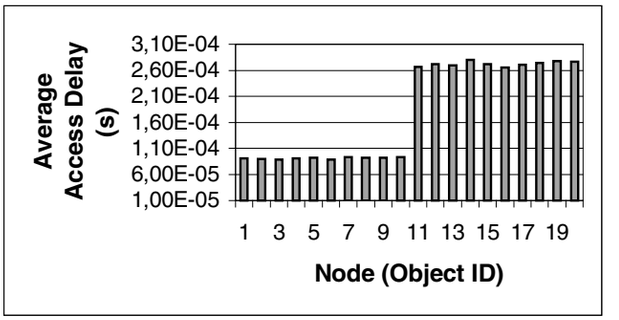


Fig. 11. RPO - Average access delay in normal workload conditions

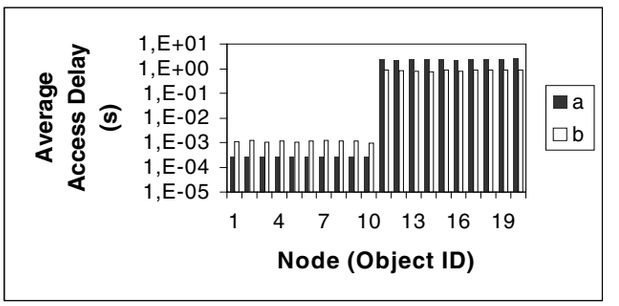


Fig. 12. RPO - Average access delay in saturation conditions with two ranges: *a*=(100, 1032) and *b*=(500, 1032)

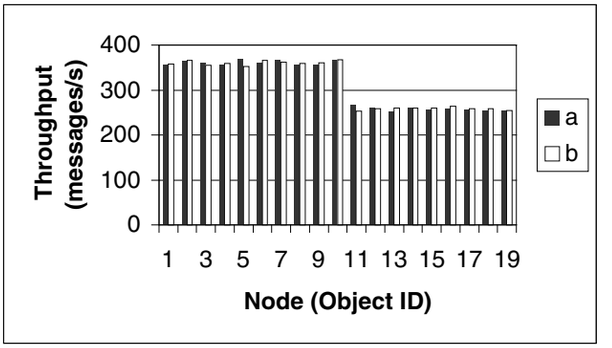


Fig. 13. RPO- Throughput in saturation conditions with: *a*=(100, 1032) and *b*=(500, 1032)

9 Conclusions

In this paper we have presented and evaluated RAP and RPO, two randomization-based approaches to enhance the management of low priority aperiodic traffic in CAN networks. The introduction of randomization makes it possible to modify the basic mechanism for access to the CAN physical channel, introducing a "controlled" degree of fairness that takes into account the requirements in terms of access delay and throughput of the different exchanged objects. That is, depending on the application requirements for aperiodic low-priority objects, either a preferential service for some class of these objects or fairness can be provided in a CAN network through a suitable choice of randomization ranges.

Here one very important difference between RAP and RPO should be pointed out. The RAP algorithm provides for the existence of different clusters of objects with different priorities. The priority levels for the various classes are disjoint, thus in the case of conflicts first class (i.e. highest priority) objects are deterministically successful over other class ones. On the contrary, in RPO the priorities are probabilistic, which means that a first class object only has greater probability of accessing the physical channel, but no statically pre-defined precedence. This means, for instance, that even in saturation conditions first class objects cannot monopolize the bandwidth, but they have only a greater probability of gaining a greater portion of it. Thus even if with the RPO algorithm first class objects are not allowed to overcome the others, they still maintain a certain degree of advantage, that is, can obtain a better quality of service in terms of bandwidth and access delay.

The proposed approaches can coexist with the CAN standard scheduling for highest priority objects (i.e. aperiodic real-time and periodic traffic), so it is possible to introduce them in CAN networks in order to manage the aperiodic bandwidth in a more flexible way.

References

1 Bosh: CAN specification. Version 2 (1991).
2. ISO - IS11898: Road Vehicles - Interchange of Digital Information - Controller Area Network for High Speed Communication (1993).
3. Allen-Bradley: DeviceNet specification. Rel. 2.0, Vol. I: Communication Model and Protocol (1997).
4. CAN in Automation (CiA): Draft standard 301, Version 3.0, CANopen, Communication Profile for Industrial Systems Based on CAL.
5. Kaiser, J., Livani, M. A.: Invocation of Real-Time Objects in a CAN Bus-System. Proc. of IEEE Intern. Symposium on Object-Oriented Real-Time Distributed Computing (May 1998).
6. Kaiser, J., Livani, M. A.: Implementing the Real-Time Publisher/Subscriber Model on the Controller Area Network (CAN). Proc. IEEE Intern. Symposium on Object-oriented Real-Time Distributed Computing (May, 1999).
7. Kim, K., Jeon, G., Hong, S., Kim, S., Kim, T.: Resource-Conscious Customization of CORBA for CAN-Based Distributed Embedded Systems. Proc. IEEE Intern. Symp. on Object-oriented Real-Time Distributed Computing (May 1999).
8. CAN in Automation (CiA) International Users and Manufacturers Group: CiA DS 201-207, CAN Application Layer for Industrial Applications.
9. Tindell, K. W., Burns, A., Wellings, A. J.: Calculating Controller Area Network (CAN) Message Response Times. Control Eng. Practice, Vol.3, N.8, (1995) 1163-1169.
10. Tindell, K.W., Hansson, H., Wellings, A. J.: Analysing Real-Time Communications: Controller Area Network. Proc. of RTSS'94, 15th IEEE Real-Time System Symposium, (1994) 259-263
11. Audsley, N.C., Burns, A.M., Richardson, F., Wellings, A. J.: Hard Real-Time Scheduling: the Deadline Monotonic Approach. Proc. of the 8th IEEE Workshop on Real-Time Operating Systems and Software, Atlanta, GA (May, 1991).
12. Kopetz, H.: Real-Time Systems Design Principles for Distributed Embedded Applications. Kluver Academic Publishers (1997).
13. Zuberi, K, Shin, K. : Non-Preemptive Scheduling of Messages on Controller Area Network for Real-Time Control Applications. Proc. of the IEEE Real-Time Systems Symposium, (1995) 240-249.
14. Zuberi, K, Shin, K: Scheduling Messages on Controller Area Network for Real-Time CIM Applications. IEEE Transactions on Robotics and Automation, Vol. 13, No. 2, (1997) 310-314.
15. Haritsa, J. R., Livny, M., Carey, M. J.: Earliest Deadline Scheduling for Real-Time Database Systems. Proc. of the IEEE Real-Time System Symposium, (1991) 232-242.
16. Cena, G., Valenzano, A: An Improved CAN Fieldbus for Industrial Applications. IEEE Trans. on Industrial Electronics, Vol.44, N.4 (Aug, 1997).
17. Cormen, T.H., Leiserson, C. E., Rivest, R.: Introduction to Algorithms. The MIT Press, Cambridge (1990).

Complex Reactive Control with Simple Synchronous Models

Reinhard Budde[1] and Axel Poigné[1]

GMD - AiS, D-53754 Sankt Augustin, Germany
reinhard.budde@gmd.de axel.poigne@gmd.de
http:ais.gmd.de/~budde http:ais.gmd.de/~ap

Abstract. Design of complex controller calls for models that are both easy to grasp and mathematically sound. They should support information hiding to enhance re-use and flexibility. The combination of the synchronous model with object-orientation promises to provide a solution. Today controller design is a multi-disciplinary effort, involving e.g. control engineers and computer scientists. The different views and styles of the disciplines involved should be smoothly integrated at a fine-grained level to support co-operation. The architecture of object-oriented synchronous models and the rationales behind it are discussed. A language supporting this style of modelling is the language sE developed by the authors.

1 Introduction and State of the Art

Controller design becomes more and more complex. Hardware is replaced by software to increase flexibility and decrease costs, the integration of previously isolated solutions is required. One cannot expect to decrease the complexity of a problem at hand by using advanced technology, but we may increase the awareness of where the problems arise. We claim that the challenge of embedded system design is not concerned with designing the static architecture of an application, for which real-time variants of the UML (e.g. RT-UML, [6]) are a suitable framework. The challenge rather is how to obtain real-time aware, understandable, verifiable, flexible and reusable designs, defining precisely, intuitively and mathematically sound the *dynamics* of an application.

Of course there are partial answers. Examples are Petri nets [12], or annotated FSM based models like FSMs with data path [7] and co-design FSMs [11]. These models are a good basis for simulation and experimentation with an application, but they fall short in precisely modelling the real-time execution behaviour of the application. Furthermore, if real-time properties are specified and used for validation and test, this can only achieved on a fairly low level of abstraction.

[1] Supported by Esprit LTR-Project 22703 (SYRF) and Esprit Project 25 514 (Crysis)

J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 19–32, 2001.

A viable abstraction of the concrete notion of time is achieved by the synchronous approach. It is based on the hypothesis of *perfect synchrony* which assumes that the reaction of a system to an external request has finished in any case before the next request arrives. Advantages of this hypothesis are: understandable and highly abstract design, and a deterministic and reproducible behaviour of the program. Languages based on this approach are Esterel and Argos, well suited to specify control-dominated applications, or Lustre that is based on data flow (see [1]). But there are three shortcomings:

- A smooth integration of the different styles supported by the different synchronous languages. This is because in a single application typically all styles are needed: control engineers often favor data-flow oriented design (to design digital filters e.g.). Computer scientists sometimes favor a design as state transformation presented as, e.g., hierarchical finite state machines.
- An established methodology to support information hiding in the sense of data abstraction is still missing for synchronous languages. Object oriented software approaches [2] meet this challenge. They can cope with the increasing complexity of embedded systems software. However, real-time constraints are usually not directly supported.
- The synchronous technology has a global system view, hence relates to centralized control systems. Control systems, however, are increasingly distributed, for quite obvious reasons of performance, fault tolerance, and distribution of sensors and actuators to different physical locations. The challenge is to find a good model for distribution retaining the benefits of synchronous programming.

We promote the idea to combine object-oriented design methodology with synchronous modelling for a comprehensive approach to embedded system design:

- Synchronous modelling for constructing reactive real-time components, and for enabling system validation by model checking.
- Object-oriented modelling in a strongly typed language, which is well suited for a robust and flexible design of complex systems.
- On top, we support a blackboard-architecture for communication between distributed synchronous components.

We offer a design environment, *sE*, which provides a smooth and well defined integration of the object-oriented and of the synchronous execution model.

In this paper, we report on the design decisions taken, and on details of implementation. First, we state requirements for the computational model. Then we give an example of an reactive object. Then the computational model and the compilation of statements is sketched. Next we demonstrate how different styles of specifying reactive behaviour are combined. Finally, we will comment on further work.

2 Modelling in *sE*

2.1 The Synchronous Model

The reactive part of an application consists of reactive objects which observe and emit signals. The behaviour of reactive objects is modelled using the synchronous approach.

A synchronous system responds to stimuli (signals) from the environment, but not continuously: the system is *reacting* or it is *idle*. Signals on input lines are collected (latched in hardware terms), but never propagated immediately to the system. The change from the idle phase to the reacting phase is effected by an activation (hardware designer would call this a clock pulse). Whether this is done periodically, using a timer subsystem of a micro controller, or depending on environment conditions is irrelevant for this discussion. The latched input signals make up the *input event* and are provided to the system.

Now the program reacts and computes the response. Signals are emitted or tested whether they are present. Subreactions are computed in parallel. The set of output signals emitted during the reaction form the *output event*. The reaction is complete, if all components of the (non-sequential) system have committed to halt. Then all signals of the output event are made available to the environment and the program is idle again.[1]

Usually output signals are connected to actuators or to display units and effect changes in the environment. These changes may affect sensors generating input signals that are latched. Then, if the clock ticks, the next reaction step is initiated.

Each such step *input*→*reaction*→*output* is called an *instant*. While reacting the system is logically disconnected from the environment, i.e. it is impossible to add input signals to the input event during a reaction. This model may appear simple and natural to a hardware-designer. But it is unusual in software design. There is no rendezvous-like concept in the synchronous model. Emitting a signal will never be blocked. Signals are broadcast, they are not consumed by the await statements. This facilitates simultaneous reactions to the same signal.

The behaviour of a system has to be *perfect*: In any case a reaction has to be finished before the next clock pulse. This property has to be proven with respect to the selected hardware and compilers.

This description reminds of the Statemate semantics[9] of Statecharts. But Statemate (and Rhapsody[8], too) are much more permissive: Behaviour is not deterministic, and cannot be reproduced, there exist different ways of communication (signals, method calls when aliasing is possible) which may interfere in a way hard to foresee (and to specify).

One may argue, that our synchronous object-oriented programming language *sE*

[1] To use the processor when i.e. the reactive part is passive, time uncritical computations may be scheduled. No assumption on the execution time of such activities can be made.

– reduces Statemate to the essentials for dependable computing,
– adds data flow constructs well-known by control engineers (without blowing up the semantics), and
– defines a non-operational, comprehensible semantics
– providing a sound foundation for applying verification technology as well as highly efficient code generators.

2.2 Object-Oriented Information Hiding

As usual in *sE* objects are defined in classes. A class defines an interface for its clients and an implementation. A class can be either a *data class* or a *reactive class*. Classes look similar to Java classes. Some restrictions to enhance readability and encapsulation apply, for instance: fields are never public, variable hiding (shadowing) is forbidden, the default visibility is private.

In general, if a conflict between flexibility/power and encapsulation arises, *sE* gives encapsulation higher priority. This is justified since we target design of dependable control applications. Take visibility of fields as an example: to allow clients to change an attribute value, you have to write a public set-method. This may be annoying, but it encourages to write assertions as pre/post conditions for a method call (design-by-contract), which allows for more rigid software reviews.

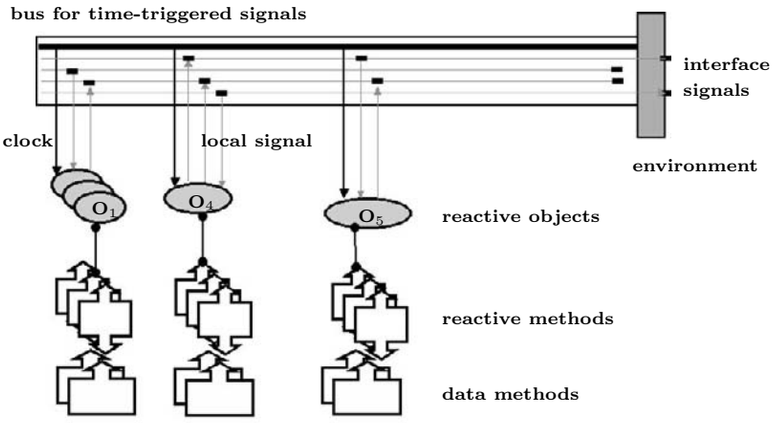


Fig. 1. The signal bus for synchronous communication

Reactive classes define an interface, which consists only of input and output signals. These signals are the *only* communication mechanism between reactive objects (see section 2.4). The implementation of reactive classes consists of

- reactive methods. These define the control related behaviour of a reactive class and are either hierarchical finite state machines (in textual or graphical format), data flow equations, or imperative statements.
- data fields and methods. These define the data related behaviour of a reactive class. Both may be used within reactive methods, but not vice versa.

Data classes define an interface, which consists of their public method. They look similar to Java classes.

For an overall picture, reactive classes may use reactive and data classes, but data classes may only use data classes. The reactive objects of any application are allocated statically forming a static tree. Data objects may be static or created dynamically. They reside exactly "below" one reactive object of the static tree of reactive objects (see figure 1) and *cannot* be shared between reactive objects. Consequently the architecture of an *sE* application – similar to many, more traditional, approaches – consists of a top layer defining the control of the application encapsulated in reactive objects, and a bottom layer of data encapsulated in objects as well.

2.3 An Example of a Reactive Class

As usually a class consists of fields and methods. The keyword "`reactive`" indicates that a method specifies reactive behaviour. Signals are a particular kind of field. We require that only signals and constructors may be public for a reactive class. Figure 2 is a rather simple example.

The constructor "`Timer`" sets the initial counter value "`latch`". The reactive method "`counter()`" behaves as follows: A timer object initially waits for the "`start`" signal. If the "`start`" signal is present, the counter is set to a start value by calling the (data) routine "`reset()`". At each "`clock`" "tick" the counter decrements until the condition "`isElapsed()`" holds. It is important to notice that "time passes" only when waiting. Reactions are considered as instantaneous: If the "`clock`" signal is present the counter decrements at the same instant as the termination condition "`isElapsed()`" is evaluated.

In *sE* it is encouraged to specify permissible scheduling between methods. Usually this expresses invariants the designer has in his or her mind and avoids time races (non-determinism). In the example only schedules are legal, in which for any reaction "`decr`" is executed before "`isElapsed`", and "`reset()`" after both). Potential time races will be detected by the *sE* compiler based on the analysis of the data flow and the state transitions. The programmer is required to specify the scheduling of method calls through *sequence clauses*, if necessary.

2.4 The Signal and the Time Race Firewall

A clear and simple model of communication was a primary goal when designing the *sE* language. It is based on the fact, that the top layer of an application are reactive objects which communicate by signals only. Signal values may never be

```
reactive class Timer {

// constructor
public Timer (int d) {
  latch = d;
};

// reactive interface
public signal input  start;
public signal input  clock;
public signal output elapsed;

// reactive implementation
private reactive rct_counter () {
  loop {
     await ?start; // wait for signal start being present
     reset();      // reset the timer count
     next;         // reaction finished for this instant
     cancel {      // decrement the counter when ..
        loop {     // .. signal clock is present
           await ?clock;
           decr();
           next;
        };                     // decrementing is cancelled, when ..
     } when (isElapsed ()) {   // .. isElapsed() is true  ..
         emit elapsed;         // .. tell that the timer elapsed
     };
  };
};

// fields
private int latch, counter;

// methods
private void    decr()      { if (counter > 0) { counter--; }; };
private boolean isElapsed() { return (counter <= 0); };

// scheduling to avoid time races, here just a consistency check ..
// .. because the schedule is completely deduced from the control flow
// .. by the compiler
sequence
  writeThenRead: decr < isElapsed;
  resetLate: decr isElapsed < reset;

}
```

Fig. 2. A simple reactive class

references to avoid hidden communication pathes by creating aliases which cross the border between reactive objects.

But one can go much further: the notion of an instant defines an interval for reactive behaviour (similar to an transaction in databases): the *sE*-compiler guarantees for every single reaction step a consistent view of the signal's presence, absence and value for all reactive objects, e.g. if one objects reads a signal value, a programmer can safely assume, that all other objects see exactly the same value. This is in sharp contrast to approaches based on threads with rendez-vous, locks or condition variables. Here sequence of declaration, system-load, priorities as used in parts elsewhere may influence the system's behaviour in a way hard to anticipate and to specify.

The consistent view of the system's state for all reactive objects is based on a write-before-read strategy: code is generated in a way, that reading a value or testing for presence is delayed until the value is emitted (and the signal is present) or emission by no reactive objects is possible *in this instant*. Section 3 explains how causality analysis can guarantee the existence of such an ordering and why this can be computed highly efficient at run-time.

The consistent view of and the clean communication discipline *between* reactive objects is called the *signal firewall*.

There is a second kind of firewall *inside* reactive objects, the *time race firewall*:

A time race occurs for instance, if during a reaction two or more methods are called, which operate on the same data, at least one of them modifies the data, and there is no well-defined schedule of the calls. Time races introduce unwanted non-determinism. Time races occur in a threaded system if data accesses are not guarded by appropriate locks. Such guards in turn may influence the overall system behaviour. In any case the scheduling is done at run-time. The *sE* compiler guarantees a well-defined schedule, furthermore the schedule is pre-computed at compile-time. This is achieved in two steps: First, methods in reactive classes must be private. So potential time races are restricted to single objects. Then it is checked, that no conflicting activation conditions for methods exist. How this is done, is explained in section 3. See also the sequence-declaration in the example of section 2.3.

2.5 Reactivity

Any *sE* program that passes the compiler is guaranteed to be reactive and deterministic. The reactive kernel may be considered as a generated scheduler for the methods of the classes involved in the application. This code then may be executed immediately on the hardware or using any hard real-time operating system.

The reactive kernel can be checked to satisfy hard real-time constraints as follows. A timing clause in a *sE*-application specifies the maximal delay the environment can tolerate between two reactions. According to the synchrony hypothesis, each reaction must terminate within this delay time. Hence one has to determine the worst case execution time of all routines with regard to all

possible reactions. For a given combination of target processor and C-Compiler. This can be checked by a tool.[5]

3 Execution Model and Compilation

3.1 The Formal Model

We define synchronous behaviour in terms of a particular kind of state machine, we refer to as *Synchronous Automata* (*Sya*).

Definition 1 *A Synchronous Automaton* $\mathcal{P}$ *is represented by a tuple*

$$(\mathcal{S}, \mathcal{R}, \mathcal{D}, \mathcal{I}, \mathcal{O}, \mathcal{P}\ , \mathcal{P}^!)$$

where

- $\mathcal{S}$ *is a set of* signals *with* $\mathcal{I}, \mathcal{O} \subseteq \mathcal{S}$ *being sets of* input *and* output *signals,* $\mathcal{R}$ *is a set of* registers*, and* $\mathcal{D}$ *is a set of* data cells*, and where*
- $\mathcal{P} : St \times 2^{\mathcal{S}} \to St$ *is a* transition function*, and* $\mathcal{P}^!\colon St \times 2^{\mathcal{S}} \to 2^{\mathcal{S}}$ *is an* output function*.* $St = 2^{\mathcal{R}} \times \mathcal{V}^{\mathcal{S}} \times \mathcal{V}^{\mathcal{D}}$ *is the set of* states*.*

Here $2^{\mathcal{X}}$ is the power set of $\mathcal{X}$, and $\mathcal{V}^{\mathcal{X}}$ is the set of valuations $v : \mathcal{X} \to \mathcal{V}$ to some values of appropriate type.

We introduce some notation for convenience:
let $\sigma = (R, v, v') \in St$. Then $\sigma(s) = v(s)$ for $s \in \mathcal{S}$, and $\sigma(d) = v'(d)$ for $d \in \mathcal{S}$. Further, we use $r \in \sigma$ to state that $r \in R$.

A synchronous automaton interacts with its environment only via the input and output signals the presence or absence, or valuation of which can be communicated. Let *inputs* and *outputs* be given by partial functions $I \subseteq \mathcal{I} \times \mathcal{V}$ and $O \subseteq \mathcal{O} \times \mathcal{V}$ (that is e.g., $(x, v), (x, v') \in I$ implies that $v = v'$).

Then a reaction step or *instant* of $\mathcal{P}$ is determined by a quadruple (I, σ, O, σ') such that (i) $\sigma' = \mathcal{P}\ (\sigma_I, E)$, and (ii) $(o, v) \in O$ iff $o \in E$ and $v = \sigma'(o)$ where σ_I equals σ except for $\sigma_I(i) = v$ if $(i, v) \in I$. The event E is the minimal event such that $P^!(\sigma_I, E) \subseteq E$ and $\{i \mid (i, v) \in I\} \subseteq E$. In words: the output and the new state is determined by the input and the old state. The fixpoint definition reflects the fact that a synchronous automaton may "emit" local signals.

The synchronous automaton is specified in terms of several kinds of functions ($\mathcal{B} = \{true, false\}$):

- for each signal $s \in \mathcal{S}$, a *presence function*

$$\delta(s)\colon 2^{\mathcal{S}} \times 2^{\mathcal{R}} \to \mathcal{B}$$

  specified by a boolean equation of the form $s = \phi$,
- for each register $r \in \mathcal{R}$, an *activation function*

$$\delta(r)\colon 2^{\mathcal{S}} \times 2^{\mathcal{R}} \to \mathcal{B}$$

  specified by a boolean equation of the form $r = \phi$,

- *signal actions* $\delta(a)\colon \mathcal{V}^{\mathcal{S}} \times \mathcal{V}^{\mathcal{D}} \to \mathcal{V}^{\mathcal{S}}$ where $a \in \mathcal{S}$,
- *void data actions* $\delta(d)\colon \mathcal{V}^{\mathcal{S}} \times \mathcal{V}^{\mathcal{D}} \to \mathcal{V}^{\mathcal{D}}$ where $d \in \mathcal{S}$, and
- *boolean data actions* $\delta(a, c)\colon \mathcal{V}^{\mathcal{S}} \times \mathcal{V}^{\mathcal{D}} \to \mathcal{B}$ where $c \in \mathcal{S}$.

$\phi \in B(\mathcal{S} + \mathcal{R})$ where $B(X)$ is the set of boolean formulas over X, i.e., formulas built from elements in X using $\wedge$, $\vee$, $\neg$ as well as $true$ and $false$.

3.2 The Translation Scheme

Rather than to go through all the formal motions, we demonstrate these ingredients in figure 3 by displaying the synchronous automaton generated by the compiler from the example as given in figure 2.

```
signals:                          registers:
g7      <- r1 or r2               r1 <- g20 or (r1 & not(beta))
g9      <- g7 or r3               r2 <- not(g15) & (g10 or (r2 & not(beta)))
g10     <- g9 & clock             r3 <- not(p2) & (g7 or r3)
a1      <- g10                    r4 <- not(g20) & (g17 or r4)
a2      <- g9
g15     <- g9 & c1
g17     <- alpha or g15           data actions:
g19     <- g17 or r4              a3 : reset ()
g20     <- g19 & start            a2 : c1 <- isElapsed ()
a3      <- g20                    a1 : decr ()
elapsed <- g15
p2      <- g15 or g10
```

Fig. 3. Synchronous automaton for class from Figure 2

The signal equations define whether a signal is present or absent (i.e. the right hand side of an equation evaluates to "`true`" resp. "`false`"). Here "`alpha`" and "`beta`" are particular *system signals* used in the translation; "`alpha`" is true only in the first instant, while "`beta`" is true only in later instants. It is required that all registers are set to "`false`" in the first instant. The void data action "`a3 : reset ()`" should be read as: if the signal "`a3`" is present then the method "`reset`" has to be executed. Void data actions are used because of their side effects on data. The boolean data action "`a2 : c1 <- isElapsed ()`" should be read as: if signal "`a2`" is present, the boolean method "`isElapsed`" has to be executed. If it evaluates to "`true`", then the signal "`c1`" is present, else absent.

The signal and register equations completely define the control layer (cf. Section 2.2). Data actions constitute the clean interface between the control and the data layer. Void data actions cause changes of data based on the state of the control layer. Control depends on data only via boolean data actions. With respect to that boolean data actions behave like input signals.

Signal and register equations may understood as presenting hardware. This is why model checking methods can be applied to pure control applications, and

even hardware optimization techniques. For instance, from signal and register equations Verilog- or BLIF-code may be generated for the VIS or SMV model checkers or the SIS hardware optimizer tool. Specification of properties either in temporal branching time logic (CTL) and Past Time Logic (PTL) are supported.

Our translation for reactive routines is denotational in that, for every syntactic constructor, a semantic routine is provided. The scheme is (slightly simplified) of the form, e.g.,

$$[[\mathcal{P}\ ;\mathcal{Q}\ ]]\ \alpha\ \beta\ \tau = ([[\mathcal{P}\ ]]\ \alpha\ \beta\ \tau) + \{\alpha'\ = \mathcal{P}.\omega\} + ([[\mathcal{Q}\ ]]\ \alpha'\ \beta\ \tau)$$

The reading is this: the sequential composition is defined by translating $\mathcal{P}$ in context of the system signals α, "first instant", β, "later instant", and τ, "preemption". Except for a set of equations and actions (as above), this translation generates a particular (synthesised0 attribute $\mathcal{P}.\omega$ that becomes *true* if the execution of $\mathcal{P}$ terminates. This is used to trigger (the first instant) of $\mathcal{Q}$ via the new system signal α'. The operator "`+`" indicates the juxtaposition of all the data generated. Here are three more examples:

$$\begin{aligned}
[[\texttt{emit s}]]\ \alpha\ \beta\ \tau &= \{s = \alpha\} \\
[[\texttt{next}]]\ \alpha\ \beta\ \tau &= \{r = \neg\tau \wedge (\alpha \vee r)\} \\
[[\texttt{cancel}\ \mathcal{P}\ \texttt{when s}]]\ \alpha\ \beta\ \tau &= ([[\mathcal{P}\ ]]\ \alpha'\ \beta\ \tau') + \{\tau' = s\}
\end{aligned}$$

(r is a newly generated register). Details of the denotational semantics are given in [13].

3.3 Causality

The signal equations are evaluated according to the write-before-read strategy: an equation may only be evaluated when all signals occurring on the right hand side are known. This defines a partial order on signals. The register equations are evaluated at the end of an instant in an arbitrary order, since their values are only used by signal equations in the next instant.

Reactive statements have a natural order of evaluation, e.g. the then-part of a conditional statement should be evaluated after its condition. This order is preserved by the translation scheme and encoded in the signal equations using auxiliary signals. Any sorting of the signal equations which preserves the partial order of signals (e.g. a topological sort) reflects the order of reactive statement. If such a sorting exists, the application is called *causally correct*. In a causally correct application all reactive objects have a consistent view of the status of all signals as required by the signal-firewall described in section 2.4. If such a sorting does not exist the compiler must reject the program. In *sE* we so far use topological sorting for causality analysis. For instance, the program "`if (not ?sig) emit sig; ;`" produces the signal equation "`sig <- alpha & not(sig)`", containing a causality cycle, hence is rejected.

Thus the signal equations have a natural "causal" order in terms write-before-read. This is less obvious for the interaction with data. Consider

```
if isElapsed then decr end
if s > 0 then emit s(5) end
```

The first equation suggests that "`isElapsed`" is evaluated before "`decr`". This is in contradiction to the sequence requirement which prescribes to opposite order of evaluation. The program is rejected by the compiler because of a causality cycle. A similar argument applies to the second example. Emittance of a signal depends on the value of the emitted signal.

In a way, such programs are (causally) over-determined. The opposite holds in case of

```
[[ reset || decr ]]
```

where the data actions are in parallel. There is no restriction on the evaluation order, hence the program may result in unacceptable non-determinism. Such a program will be rejected as a time race. The programmer may resolve a time race by a sequence statements.

The upshot is that all *sE*-programs contain reactive objects operating in parallel, may use the parallel operator, but are guaranteed to be deterministically scheduled. They show reproducible behaviour.

4 The Combination of Styles

The engineering of embedded systems, let it be for control or signal processing, is concerned with several types of models of quite different nature. A taxonomy of of models is given in Figure 4; with *time* and *state* as coordinates, either being discrete or continuous.

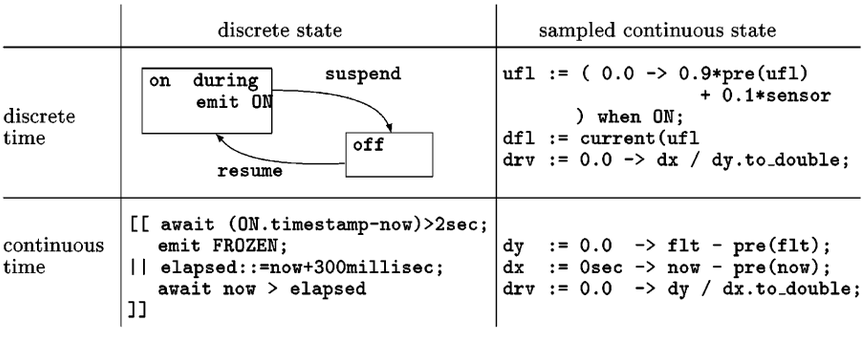


Fig. 4. Styles in controller design

Though it may sometimes be possible to model parts of a system within one single square, e.g., analog controllers in terms of continuous time and state,

more often even parts of a system can be adequately modelled only by combining different aspects, e.g., both discrete and continuous time is needed for designing a compact disc player, or discrete change of state may be needed for a sensor/actuator system to model major changes of mode of operation.

All synchronous formalisms assume time to be discrete but differ with regard to the modelling of state. Available synchronous formalisms are either imperative supporting discrete state transitions or are declarative supporting continuous state transition where subsystems may run on a wide variety of clock rates. Hence a combination of these formalisms will allow to explore the full design space of discrete time models in a uniform way.

It is an outstanding feature of *sE* that it supports all these models in a uniform and efficient way. The continuous time dimension is interpreted in a rather specific way in that we relate it to real time; the signal "`now`" provides a time stamp in terms of system time at the beginning of an instant (presently in micro seconds). hence the difference "`now - pre(now)`" defines the real time measured between two instants, i.e. dx.

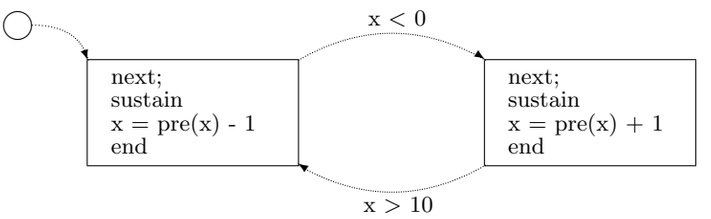


Fig. 5. Data flow in finite state machines

The unification of the idioms is achieved on the level of signals. As mentioned earlier, signals have a presence and a value. Values may only changed if a signal is present. There are, however, different mechanisms for specifying the presence. Signals may either be emitted within a control flow by a statement such as

$$\texttt{emit s(e}_1\texttt{,...,e}_n\texttt{)}$$

where "`s`" is a signal, and "$\texttt{e}_1$",...,"$\texttt{e}_n$" is a vector of expressions (possibly of length 0, then we speak of *pure signals*). Such signals are called *sporadic* or *episodic*.

On the other hand signals may be sampled continuously. We speak of *periodic* signals or *flows*. The periodicity of the signal is specified in its declaration. The declaration

```
public signal (double value) input sampled at (threshold > 1.2) ad_value
```

states that signal "`ad_value`" is present whenever the expression after "`at`" is true. We speak of the *clock* of a periodic signal if referring to the Boolean expression. A signal is constrained by a *flow definition*, e.g.

```
x = 0 -> pre(x) + 1;
```

We do not have the space to go into detail here, but just want to remark that the styles may be mixed quite freely. Definitions of periodic signals may be, e.g., embedded in a graphically presented discrete time/discrete state model as in Figure 5. This is a common case: the state model expresses different modes of an application, the flow defines different periodic behaviour for each mode (e.g. different filters).
Note that only one flow definition should be apply at every instant to avoid non-determinism, hence the "`next`" which delays evaluation by one instant.

5 Conclusion and Outlook

We have sketched a mathematically well defined and simple model for object-oriented design of embedded systems based on the synchronous approach. We claim the smooth fine-grained integration of different styles of reactive behaviour as being a highlight of our approach.

We see a second highlight in the carefully crafted interaction of the object-oriented programming paradigm with reactive behaviour: communication of reactive objects is restricted to signal broadcast constraining time races of data operations to objects. The compiler enforces resolving of time races resulting in deterministic behaviour. On application level, determinism is guaranteed by causality analysis on the level of signals, by standard techniques of synchronous languages. These strict and well defined interfaces between objects and between reactive routines and data routines may be considered as "firewalls" or encapsulations which provide the programmer with a good degree of freedom of design within these confines.

Complex embedded systems often consists of many (synchronous) components, possibly running at different frequencies (as, e.g., motor actuators, sensors, analysis of video data, and planning modules in a robot), possibly running on different processors, communicating via, e.g., a field bus.

We have proposed (and experiment with) a non-blocking communication via blackboards (see [3,4]) with only state information (and not events) being communicated. It supports a defensive style of programming: each component has to take care of all eventual behaviours of its environment. Time stamps are used to detect the ageing of received messages. This design of the interior of a controller is in line with time-triggered protocols between controllers.[10]

Adding and removing synchronous components is another interesting topic. To maintain the compact and highly efficient code, we experiment with run-time modification of the signal- and register-equations described earlier. The compiler has to guarantee in any case, that creating and deleting reactive objects will not introduce causality cycles or non-determinism. This work will be completed in the CRISYS project.

References

1. A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9), 1991.
2. Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, Redwood City, 2 edition, 1994.
3. R. Budde, M. Pinna, and A. Poigne. Coordiantion of synchronous languages. In P. Ciancarini and A.L. Wolf, editors, *3rd Int'l Conference on Coordination Languages and Models*, volume 1594 of *LNCS*, pages 103–107. Springer, 1999.
4. R. Budde and A. Poigne. On the synthesis of distributed synchronous processes. In *MOVEP'2k: Modelling and Verification of parallel Processes*, LNCS. Springer, 2000.
5. Reinhard Budde, P. G. Plöger, and Karl H. Sylla. A synchronous object oriented design flow for embedded applications. In *Proc. 2nd Int'l Forum on Design Languages*, 1999.
6. Bruce Powel Douglass. *Real-time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1998.
7. Daniel D. Gajsk, Nikil Dutt, Allen Wu, and Steve Lin. *High-Level Synthesis : Introduction to Chip and System Design*. Kluwer Academic Pub, January 1992.
8. D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, pages 31–42, 1997.
9. David Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACMTSEM: ACM Transactions on Software Engineering and Methodology*, 5, 1996.
10. H. Kopetz. *Real-Time Systems — Design Principles for Distributed Embedded Applications*, volume 395. Kluwer International Series in Engineering and Computer Science, 1997.
11. M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavango, and A. Sangiovanni-Vincentelli. Synthesis of mixed software-hardware implementation from CFSM specifications. In *Proceeding of International Workshop on Hardware-Software Codesign*, Oct. 1993.
12. J. L. Peterson. *Petri Net Theory and the Modelling of Systems*. Prentice-Hall, Englewoods Cliffs, New Jersey, 1981.
13. A. Poigne and L. Holenderski. On the combination of synchronous languages. In W.P. de Roever, editor, *Workshop on Compositionality: The Significant Difference*, volume 1536 of *LNCS*, pages 490–514. Springer, 1998.

Optimistic Secure Real-Time Concurrency Control Using Multiple Data Version

Byeong-Soo Jeong, Daeho Kim, and Sungyoung Lee

School of Electronics & Information, Kyung Hee University
Kyung Ki Do, Young In Si, Seo Cheon Ri 1, Korea
{jeong, tonkey, sylee}@nms.kyunghee.ac.kr

Abstract. In many real time applications, security is an important requirement, since the system maintains sensitive information to be shared by multiple users with different security levels. A secure real-time database system must satisfy not only logical data consistency but also the timing constraints and security requirements associated with transactions. Even though an optimistic concurrency control method outperforms locking based methods in firm real-time database systems, in which late transactions are immediately discarded, existing secure real-time concurrency control methods are mostly based on locking. In this paper, we propose a new optimistic concurrency control protocol for secure real-time database systems. We also compare the performance characteristics of our protocol with locking based methods while varying workloads. The results show that optimistic concurrency control performs well over a wide range of system loading and resource availability conditions.,

1 Introduction

Depending on the application, database systems need to satisfy some additional requirements over and above logical data consistency. In real-time applications such as control systems, transactions have explicit timing constraints that they should meet while maintaining data consistency. In secure applications such as military applications, data and transactions are classified into different levels of security and the high security level information should be prevented from flowing into lower security levels. Because of these additional requirements, conventional concurrency control techniques cannot be used directly in such advanced database applications.

In real-time database systems, the deadline of a transaction is combined with its time-critical priority and system resources are scheduled in favor of the high priority transaction so that deadline-missing transactions are minimized. If we use conventional concurrency control mechanisms in a real-time database, priority inversion problems can occur due to shared data access [1].

* This work was supported by International collaborative Research Program (TRP-9802-6) sponsored by Ministry of Information and Communication of Korea

J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 33-47, 2001.

In secure databases, a low security level transaction can be aborted or delayed by a high security level transaction due to shared data access. In secure databases, a low security level transaction can be aborted or delayed by a high security level transaction due to shared data access. Thus, by aborting low-security level transactions in the predetermined manner, high-security level information can be indirectly transferred to the lower security level, in what is called a covert channel [8].

Recently, there is increasing need for supporting applications which have timing constraints while managing sensitive data in advanced database systems such as military command control systems and stock information systems. Thus, we need to integrate security requirements into real-time database systems. There has been considerable research and performance study on the concurrency control protocols for real-time databases and secure databases. But secure real-time concurrency control protocols are rarely presented. To our knowledge, SRT-2PL (Secure Real-Time Two-Phase Locking) [13] and PSMVL (Priority-driven Secure Multi-Version Locking Protocol) [14] are the only examples able to solve both real-time and secure requirements together.

As we can tell by their names, these protocols are based on locking protocols. Performance studies of concurrency control methods in real-time environments have shown that an optimistic approach can outperform locking based methods [5, 6]. A key reason for this result is that the optimistic method, due to its validation stage conflict resolution, ensures that eventually discarded transactions do not restart other transactions, unlike the locking approach in which soon-to-be-discarded transactions may restart other transactions. Even though an optimistic concurrency control method outperforms locking in firm real-time database systems, in which late transactions are immediately discarded, existing secure real-time concurrency control methods are mostly based on locking.

In this paper, we propose a new concurrency control protocol based on an optimistic method for secure real-time database systems. The proposed protocol solves the conflicts between real-time constraints and security requirements by maintaining multiple data versions and ensures serializability by appropriately marking conflicting transactions and letting them continuously proceed with the correct data versions. Our protocol eliminates the priority inversion and covert channel problems. The schedules produced by our protocol also ensure serializability. We devise data structures and several rules that can maintain multiple data versions efficiently. We also compare the performance characteristics of our protocol with locking based methods from the viewpoint of deadline-missing transactions and the degree of wasted restarts under varying workload conditions.

The rest of the paper is organized as follows. In section 2, some related work is reviewed. In section 3, we briefly describe a secure real-time database model and present our secure real-time protocol. In section 4, we discuss the logical correctness of the proposed protocol. The results of the simulation experiment are described by comparing it with locking based methods in section 5. Finally, we conclude our study in section 6.

2 Related Research

In the early stage, real-time concurrency control protocols and multi-level secure concurrency control protocols have been studied independently. The former has been researched on the basis of locking strategies or optimistic methods, including priority based scheduling approaches. In the case of using locking strategies, typical examples are Wait Promote, High Priority, and Conditional Restart algorithms [1]. These algorithms aim to schedule transactions without causing priority inversion problems. OPT-SACRIFICE, OPT-WAIT, and WAIT-50 [5] are several alternatives that exploit optimistic techniques in a real-time database environment. The optimistic protocols use priority information in the resolution of data conflicts, that is, they resolve data conflicts always in favor of higher priority transactions. In a transaction's validation stage, if there are conflicting higher priority transactions, it should be restarted (sacrificed) or blocked (waited) until higher priority transactions commit. WAIT-50 is a hybrid algorithm that controls the amount of waiting based on transaction conflict states.

Multi-level secure concurrency control protocol, whose research is mainly based on the Bell-LaPadula [2] security model, aims to maintain noninterference among transactions with different security levels. In a secure database, problems occur when there are data conflicts between different security level transactions. If we allow a low security level transaction to be aborted or blocked by a high security level transaction during the resolution of data conflicts, a covert channel can be made. The secure concurrency control protocols have been researched mainly on the basis of using multiple data versions or a single version. In the case of using multiple data versions, when different security level transactions request same data item simultaneously, they use different versions to avoid interference [8]. A major concern of these protocols is how to select a correct version in order to ensure serializability and how to maintain multiple versions considering storage overhead. In the case of using a single version, a low level transaction should not be blocked or aborted by a high level transaction. The secure 2-phase locking protocol proposed by Son [16] tries to simulate execution of basic 2PL without the blocking of low level transactions by high level transactions. Blocking occurs when a low level transaction (say $\boldsymbol{T}_L$) asks for a write operation on a data item that is already read-locked by a high level transaction (say $\boldsymbol{T}_H$). The secure 2PL protocol allocates a virtual lock to $\boldsymbol{T}_L$ and allows it to execute the write operation, without delay, on the local area. Once the lock is released by $\boldsymbol{T}_H$, the virtual lock is changed to a real lock, and an actual write operation is executed on the database.

Recently, some integrated protocols that satisfy real-time constraints and security requirements together have been proposed. Mukkamaala and Son [13] have introduced SRT-2PL, which prevents covert channels by maintaining the property of noninterference among transactions with different security levels. Since a covert channel arises when a high level transaction aborts or delays a low level transaction, the SRT-2PL removes possible conflicts by separating transactions with an access class identical to the data object from higher access class transactions. This is accomplished by maintaining two copies (a primary copy and a secondary copy) of each data object. The primary copy of an object is accessed (read/write) by transactions at the same security level as the data object. The secondary copy is accessed (read) by transactions at a security level higher than the data object. The

update of the primary copy is carried out on the secondary copy through the queue that contains each update of the primary copy in the same order of update.

Park et. al. [14] propose another secure real-time protocol based on a multi-version 2-phase locking protocol. It introduces a new concept of serializability, what is called F (First-read) serializability [15], which is more general than the definition of 1-copy serializability. F-serial eliminates the limitation that transactions must read the most recent versions and provides the boundary of each version that a transaction can read when the transaction does not read the most recent version. Thus, it can provide a higher degree of concurrency than MV2PL (Multi-Version 2-Phase Locking) which is based on 1-copy serial. The other works which need to be mentioned, are [4] and [10]. In [4], a novel dual approach is proposed that allows a real-time database system to simultaneously use different concurrency control mechanisms for guaranteeing security and for improving real-time performance. In [10], a new optimistic concurrency control algorithm is presented which can avoid unnecessary restarts by adjusting serialization order dynamically.

3 Optimistic Secure Real-Time Concurrency Control

Performance studies of concurrency control algorithms for conventional database systems have shown that locking protocols outperform optimistic techniques under most operating circumstances. However, in firm real-time database systems where late transactions are immediately discarded, an optimistic technique is considered as more advantageous from the viewpoint of real-time performance, i.e., meeting timing constraints instead of transaction's response time [5, 6]. Figure 1 shows one example of such scheduling advantage. Let's consider the following schedule of transactions, ***T1*** and ***T2*** that require 5 execution time units and have deadlines, time unit 4 and 6 respectively. In the case of lock-based protocols, ***T2*** cannot meet deadline (time 6) due to blocking of soon-to-be-discarded transaction, ***T1***. On the contrary, in optimistic protocol, ***T2*** can meet the deadline and successfully commit since ***T2***'s validation happens after ***T1***'s abortion. With such advantages in mind, we propose a new secure real-time concurrency control protocol based on optimistic approach.

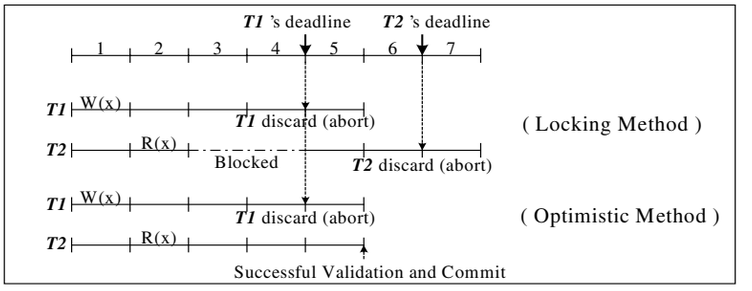


Fig. 1. Transaction Scheduling

In our secure real-time database model, each transaction (and data) has its own security level and priority value assigned by considering the transaction's deadline (we assume that the priority value is unique among transactions). A basic principle of our protocol is that if data conflict occurs in the transaction's validation phase, conflict is resolved to meet real-time and security requirements. In order to meet the real-time requirement and avoid covert channels, lower priority transaction and high security level transactions should be restarted (aborted) or delayed (waited) during conflict resolution. Data conflict, which is considered in the forward validation, arises from the data set written by the validating transaction and the data set read by currently running transactions. When we solve real-time and security requirements in single scheme, the problem occurs in the case that validating transaction has low priority value and low security level and conflicting active transaction has high priority value and high security level (after now, we call this case as LL-HH conflict). If we abort a validating transaction for a real-time requirement, it can cause a security violation. Aborting active transactions violates the real-time requirement. This case happens when a low priority transaction, which has written on a data item already read (i.e., read-down) by high priority transactions, enters the validation phase. Our proposed protocol solves these conflicts between real-time and security requirements by using multiple versions of data objects.

In the proposed protocol, when a data conflict of such a type happens, we commit the validating transaction and also let the active transaction continue to progress by selecting correct versions of the data object in order to ensure serializable execution. In the transaction's validation phase, if such a case happens, the conflicting active transactions are marked with a timestamp, MTS(Marked Time Stamp) at this validation point and inserted into an MTL (Marked Transaction List) with the transaction identifier and timestamp value. After a transaction is marked, a read operation of the marked transaction is executed on the correct data version by comparing this timestamp (MTS) with that of the data version. In what follows, we now turn to describe our protocol in more detail by presenting several rules and data structures.

In the proposed protocol, transaction read/write operations are performed according to the following rules.

[Rule 1] A transaction's validation is processed sequentially (one at a time) and at this time a unique timestamp value is given. The timestamp value is the logical time value that indicates a logical order of transaction validation.

[Rule 2] At first, every write operation is performed in the transaction's local workspace. After the validation is successfully finished, the write operation is reflected in the database. At this time, a new data version is generated and the data WTS(Write Time Stamp) is recorded as the timestamp of the validation point.

[Rule 3] In the transaction's validation phase, if the LL-HH case happens, active transactions are marked with the timestamp value of the validation point, we call it an MTS (Marked Time Stamp), and inserted into the MTL (Marked Transaction List) with MTS value and transaction identifier.

[Rule 4] Read operations of marked transactions use the most recent version of data $\boldsymbol{x}$ where $WTS(\boldsymbol{x}) < MTS(\boldsymbol{T})$.

[Rule 5] Read operations of unmarked transactions always use the most recent data versions.
[Rule 6] At the time of transactions' read-downs, which means read operations by transactions at a security level higher than the data objects, a transaction identifier is inserted into the RDTL (Read-Down Transaction List) of the corresponding data version.
[Rule 7] When the marked transaction is committed or restarted (aborted), the corresponding node of that transaction in the MTL is deleted.
[Rule 8] When the transaction is committed or restarted (aborted), the corresponding transaction identifier is deleted from the RDTL of the data version if it has performed read-down operations on that version.
[Rule 9] In the case where the MTL changes (the case where a marked transaction is committed or restarted), all data versions but one where the WTS value is smaller than the smallest MTS value in the MTL are deleted.
[Rule 10] If the RDTL of the data version is empty and it does not violate [***Rule 9***], the corresponding data versions are deleted.

In our protocol, the execution of a transaction consists of three phases: read, validation, and write phase. The read phase process can be described by the following procedure.

```
read_phase(Ta)
{
        foreach Di (i = 1, 2, ..., m) in RS(Ta) {
            if (Ta is marked)
                        select the version of Di where WTS(Di) < MTS(Ta)
                        read that version
            else        read the most recent version of Di
        }
        foreach Dj (j = 1, 2, ..., n) in WS(Ta)
            write Dj in local workspace
}
```

After the read phase, a transaction goes through the validation as in the forward validation procedure. If data conflicts occur during the validation, conflict resolution is executed as follows.

```
conflict_resoution(Tv)
{   // P(T): Priority of Transaction, T    L(T): Security Level of Transaction, T
   case 1: P(Tv) > P(Ta) and L(Tv) = L(Ta)
            abort(Ta);    commit(Tv);
   case 2: P(Tv) > P(Ta) and L(Tv) < L(Ta)
           abort(Ta);    commit(Tv);
   case 3: P(Tv) < P(Ta) and L(Tv) = L(Ta)
            abort(Tv);
   case 4: P(Tv) < P(Ta) and L(Tv) < L(Ta)
            mark(Ta);   commit(Tv);
    }
```

Once the validation of a transaction is successful, the transaction's updates are copied from the local workspace into the database in the write phase. At this time, a new version of the data object is generated with the WTS as the timestamp value of the validation point.

In order to describe our protocol more clearly, we explain step-by-step operations according to the above procedures for the sample history in Table 1. Figure 2 shows data structures used in the protocol and how it changes while executing transactions. In Figure 2, data structures enclosed by a solid line represent the initial state of data versions and MTL, and the update of data versions and MTL is represented by a dotted line. In the sample history of Table 1, until the validation of T_2 begins, transactions' read/write operations are performed without any interference. *r(x)* and *r(y)* of T_1 read the most recent data version, *x=30* and *y=20* respectively. *r(y)* of T_2 also reads *y=20*. At this moment, since *r(y)* of T_1 is a read-down operation, T_1's *ID* is inserted in the RDTL of data version (*y=20*). *w(x), w(y)*, and *w(z)* of T_1 and T_2 are executed on each transaction's local workspaces.

Table 1. Sample History

Time / *Trans.*	…	10	11	12	13	14	15	16	17	18	…
T1		*r(x)*			*w(x)*		*r(y)*		*r(z)*	***V***	…
T2			*r(x)*	*w(y)*		*w(z)*		***V***	…		

$P(T_1) > P(T_2)$ $P(T_i)$: T_i's priority

$L(T_1) = L(x) > L(T_2) = L(y) = L(z)$ $L(T_i)$: T_i's security level

V : validation $L(x)$: Security level of data *x*

During the validation of T_2 at time 16 in Table 1, LL-HH conflict happens due to data *y*. Therefore, T_1 is marked (See Figure 2 (a)) according to the above rules and after this, T_2 is committed successfully reflecting *w(y)* and *w(z)* to the database (at this moment, new versions of data *y* and *z* are generated). At time 17, since T_1 is marked by T_2, *r(z)* of T_1 selects data version, *z=4* (where $WTS(z) < MTS(T_1)$) instead of using data version, *z=10* which is the most recent version of *z*. After the successful validation of T_1, T_1 is committed. In consequence of T_1's commitment, a new version of *x* is generated and the node of T_1 in MTL is deleted. If the smallest MTS value in the MTL is changed, the version management procedure is executed according to [***Rule 9***]. For example, if T_3 is deleted from MTL (deletion of T_3 occurs when T_3 is committed or aborted), the version management procedure deletes the first version of *x* (value=*10*, WTS=*2*) since x already has a version (value=*30*, WTS=*7*) whose WTS value is smaller than the smallest MTS value, *8* in MTL.

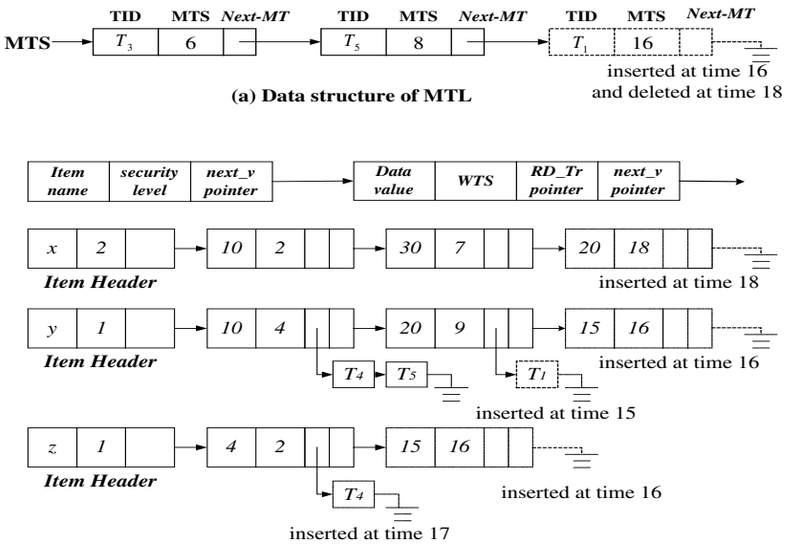


Fig. 2. Data Structures

4 The Correctness of the Protocol

In this section, we prove the correctness of the proposed protocol. To prove logical correctness (i.e., serializability), we use the simple definitions of a history and a serialization graph. In order to show that our protocol satisfies real-time and security requirements, we briefly state the results less formally in **[Lemma 3]** and **[Lemma 4]**.

[Lemma 1] All committed transactions at a single security level are serializable.

Proof : In the case where every transaction has the same security level, the execution of transactions is the same as in the existing real-time optimistic protocol of [5] since the LL-HH conflict does not occur. Thus, we know that all committed transactions are serializable as in [5]. The serial order of committed transactions is the same as the validation order of transactions.

[Lemma 2] All committed transactions (independent of their security level) are serializable.

Proof : By **[Lemma 1]**, all committed transactions at a single level are serializable. Thus, we only need to prove that the committed transactions with different security levels are also serializable. Consider levels **L** and **L′** where **L > L′.** Since transactions

at each level are serialized (by **[Lemma 1]**), let let $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_{m-1} \rightarrow T_m$ be the serialization order at level **L** and $t_1 \rightarrow t_2 \rightarrow ... \rightarrow t_{n-1} \rightarrow t_n$ be the order at level **L′** of committed transactions. Suppose, by way of contradiction, that there is no serialization order among all these transactions. Then there is a cycle in the corresponding serialization graph so that one of the following two cases is possible (Let T_i and T_k be high security level transactions and t_j and t_k be low security level transactions):

Case 1) $T_i \rightarrow t_j \rightarrow t_k \rightarrow T_i$: Since only read-down operations are allowed for higher level transactions on lower level data objects, $T_i \rightarrow t_j$ implies that there is a data object x (at level **L′**) that was read (read-down) by T_i and modified later by t_j. In our protocol, this can happen in two ways, (**I**) T_i is committed before t_j through a successful validation and (**II**) T_i is committed after t_j's commitment. In the case of (**II**), T_i is marked during the validation of t_j. Similarly, $t_k \rightarrow T_i$ implies that there is a data object y (at level **L′**) that was written by t_k and later read by T_i. In the case of (**I**), since T_i's timestamp (at the validation point) is smaller than t_j's timestamp and the timestamp of t_j is smaller than that of t_k (by $t_j \rightarrow t_k$), $t_k \rightarrow T_i$ can not happen. Also in the case of (**II**), T_i's read operation uses only the data version whose WTS value is smaller than MTS(T_i) and the WTS value of the data version which is written by t_k, is always larger than MTS(T_i) (since timestamp(t_k) > timestamp(t_j) and timestamp(t_j) = MTS(T_i)). Thus, neither can $t_k \rightarrow T_i$ happen in the case of (**II**). Therefore, the cyclic serialization graph of $T_i \rightarrow t_j \rightarrow t_k \rightarrow T_i$ cannot occur in our protocol.

Case 2) $t_i \rightarrow T_j \rightarrow T_k \rightarrow t_i$: This case also can be similarly contradicted as in the above case. In the cyclic serialization graph, $t_i \rightarrow T_j$ implies that there is a data object x (at level **L′**) that was modified by t_i and later was read by T_j. This means that t_i is committed before T_j. Similarly, $T_k \rightarrow t_i$ implies that there is a data object y (at level **L′**) that was read by T_k and later modified by t_i. Figure 3 shows the schedule of this case. In the cycle, $T_j \rightarrow T_k$ implies that T_k reads a data object z that was modified by T_j. But, as in Figure 3, T_k is marked at the time of t_i's validation and after T_k reads only the data version whose WTS value is smaller than the timestamp of t_i. Thus, $T_j \rightarrow T_k$ cannot occur in our protocol. Therefore, the cyclic serialization graph of $t_i \rightarrow T_j \rightarrow T_k \rightarrow t_i$ also cannot happen.

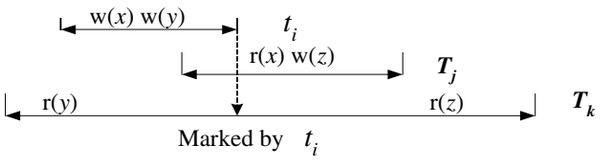


Fig. 3. Example of the Schedule

The same proof can be extended even when transactions from more than two levels are considered. Thus, all committed transactions by the proposed protocol are serializable. ■

[Lemma 3] In our protocol, low security level transactions are not delayed (Delay Secure) or aborted (Recovery Secure) by data conflicts with higher security level transactions. Data updates by higher security level transactions are not seen by lower security level transactions (Value Secure): (the necessary and sufficient conditions for a secure database).

Proof : Since our protocol is based on an optimistic method, transaction's delay, which is caused by blocking in locking protocol, does not occur. Also by **[*Rule 3*]** and **[*Rule 4*]**, transaction abort (restart) by higher security level transactions does not occur. Neither does transaction write-down occur because of the Bell-LaPadula security model. ■

[Lemma 4] High priority transactions are neither aborted (restarted) nor delayed (blocked) by low-priority transactions due to data contention on low security level data objects.

Proof : Same as the proof of **[Lemma 3]**. ■

5 Performance Evaluation

In this section, we describe the structure and details of the simulation model and experimental environment that were used to evaluate the performance of our protocol. We also present performance results from our experimental comparisons with locking based protocols.

5.1 Simulation Model

Figure 4 illustrates the simulation queueing model designed for the experiment. Each transaction, which consists of a sequence of page read and write operations, is generated from the transaction generator and then inserted into the CPU queue. In our simulation, transactions arrive in a Poisson stream, i.e. their inter-arrival times are exponentially distributed. We varied the mean transaction arrival rate in order to experiment with different (heavy or light) transaction workloads. When a transaction in the CPU queue is selected by its priority which is calculated from deadline information, it must go through the given concurrency control protocol to obtain a data access permission. If data access is granted, the transaction proceeds to perform the data operations, which consist of disk access and CPU computations. If the data access request is denied, the transaction will be placed into a blocked queue. A

transaction, which is to be aborted by the protocol, can be restarted or discarded if it cannot meet the deadline. Transactions that have already missed their deadlines are immediately discarded from CPU queue.

Table 2 summarizes the key parameters that are used in the simulation. The database itself is modeled as a collection of data pages in disks. The slack value is used to calculate the deadline according to the following formula: ***deadline = arrival_time + slack * transaction_size * execution_time***. This deadline information is also used to assign the priority of each transaction. We used the EDF (Earliest Deadline First) algorithm for the priority assignment of transactions, varying the slack value for the purpose of representing different real-time environments. The ***security_level*** of a transaction is randomly given at its generation and data objects are initially classified into different ***security_levels***. The ***transaction_size*** represents the number of read/write operations in a transaction and it is also varied in our simulation. The use of a database buffer pool is simulated using probability. When a transaction attempts to read a data page, the system determines whether the page is in memory or disk using the probability, ***Buffer_Hit***. The ***execution_time***, which is used to calculate the deadline, is obtained as follows: ***execution_time = CPU_time + (1 - Buffer_Hit) * Disk_time*.**

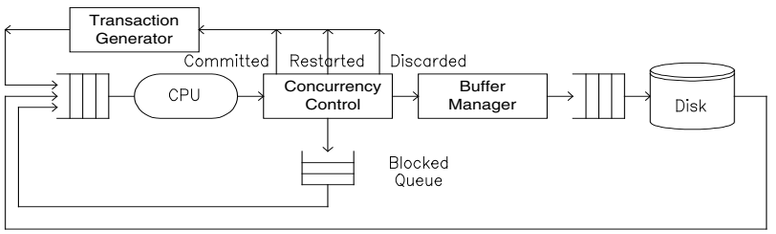


Fig. 4. The Simulation Queueing Model

Table 2. Simulation Parameters

Parameters	Value
Database Size	100
Slack Value	1.5~4.5
Security Level	3
Transaction Size	5~20
CPR Time	3.0 ms
Disk Time	25 ms
Buffer Hit	0.7
Transaction Arrival Rate	5~30 tr./sec

5.2 Experiments and Results

For experiments intended to compare our optimistic protocol with locking protocols, we implemented PSMVL[14] which is a lock-based real-time secure protocol. Since it uses multiple versions of data in order to satisfy real-time and security requirements as we do, we can examine different behaviors between them. As mentioned in [10], it is difficult to compare the performance of an optimistic protocol with that of a locking protocol due to the significant difference in their implementation. In our simulation, we only consider the impact of data contention in two schemes while assuming that the processing overhead of each protocol is the same. The simulation program was written by using CSIM libraries on the Sun UltraSparc workstation.

As for the primary performance metrics, we used ***Miss Ratio,*** which is calculated as follows: ***Miss Ratio = 100 * (# of discarded transactions / # of transactions arrived***) and ***Restart Ratio,*** which measures the average ratio of transaction restarts. We measured these metrics while varying system workloads, transaction size, and deadline tightness (by using different slack values). Additionally, we measured other statistical information, including average transaction blocking time and unnecessary restart/blocking time caused by soon-to-be-discarded transactions.

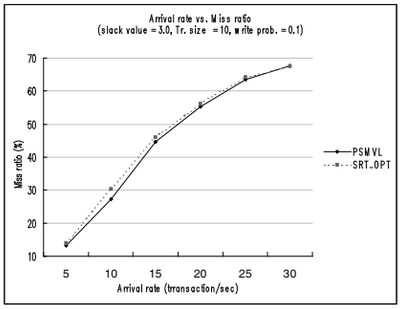


Fig. 5. Miss Ratio vs. Arrival Rate (Write Probability = 0.1)

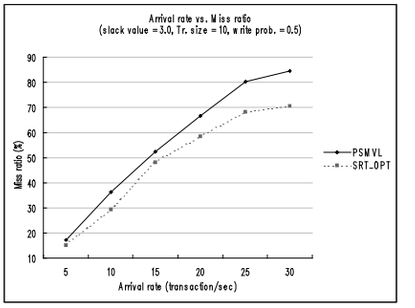


Fig. 6. Miss Ratio vs. Arrival Rate (Write Probability = 0.5)

Figures 5 and 6 (in the graph SRT_OPT represent our protocol, Secure Real-Time Optimistic protocol) show the miss ratios of two concurrency control schemes under different degrees of data contention (write probability = 0.1 and 0.5, respectively) while varying system workloads (by varying transaction arrival rate). As we can see from Figure 5, in the case where write probability = 0.1 (i.e., the operations in a transaction are mostly read) the performance of the two schemes is almost the same. The reason is that the two schemes behave similarly since there is not much data contention. However, in the case where data contention increases with high write probability and heavy workloads as in Figure 6, the performance difference becomes bigger. This is due to the fact that in the locking scheme many restarts (aborts) are made unnecessarily by soon-to-be-discarded transactions.

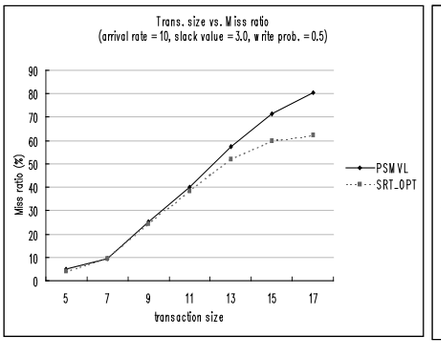


Fig. 7. Miss Ratio vs. Transaction Size

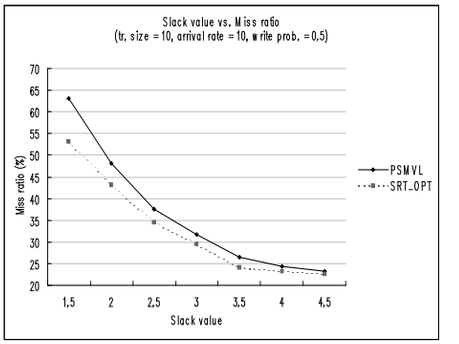


Fig. 8. Miss Ratio vs. Transaction Size

The same reasoning can be applied in Figure 7. That is, if the transaction size increases, the possibility of data contention becomes larger. Figure 8 shows the miss ratio with different slack values (deadline tightness). From Figure 8, we can see that if the transaction has enough slack time, most transactions can meet the deadline in both schemes by restarting conflicting transactions.

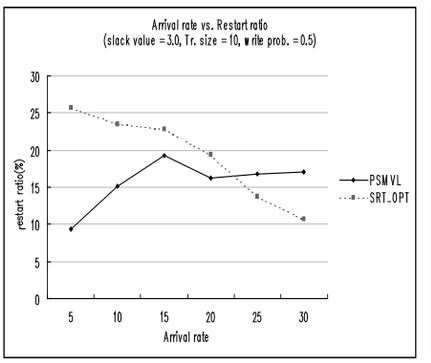


Fig. 9. Restart Ratio vs. Arrival Rate

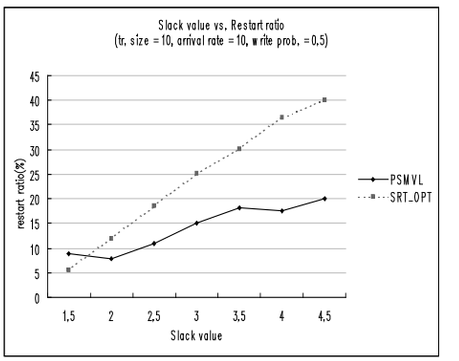


Fig. 10. Restart Ratio vs. Slack Value

Figures 9 and 10 show the restart ratio with different arrival rate and slack values, respectively. The restart ratio represents the average restart percentage of each transaction during the simulation. Actually, this metric does not give a meaningful comparison between the two schemes because the data conflicts are resolved differently in the two schemes. That is, in an optimistic approach, conflict resolution is accomplished only by restarting (aborting) one of the conflicting transactions. On the other hand, the locking based method resolves conflicts by restarting or sometimes by blocking a transaction. However, we can see that the restart ratio decreases as the system workload becomes heavier in Figure 8. The reason is that the chance of restarting becomes low as a heavier workload yields more deadline missing transactions. Figure 10 also shows similar reasoning. Enough slack time gives more chance of successful completion by restarting transactions more frequently.

6 Conclusion

In this paper, we have presented a new secure real-time concurrency control scheme based on an optimistic approach. Our protocol solves the conflict between real-time constraints and security requirements by maintaining multiple data versions, and it ensures serializability by appropriately marking conflicting transactions and letting them continuously proceed with the correct data versions. To evaluate its performance characteristics, we compared it with the existing locking based method, PSMVL. Even though we did not consider the protocol overhead of each scheme in detail, we could compare the effect of data contention between two schemes. Our experiments show that an optimistic approach gives better performance than the locking scheme in the case of high data contention because the forward validation of an optimistic method can reduce unnecessary restarts to a higher degree than the locking method can.

7 Reference

[1] R. K. Abbott and H. Garcia-Molina , "Scheduling Real-Time Transactions : A Performance Evaluation", In *ACM Transactions on Database Systems*, 17, pp513-560, 1992.

[2] D. E. Bell and L. J. LaPadula, "Secure Computer Systems : Unified Exposition and Multics Interpretation", *The Mitre Corp*., 1976.

[3] Rasikan David, Sang H. Son and Ravi Mukkamala, "Supporting Security Requirements in Multilevel Real-Time Databases", In *IEEE Symposium on Security and Privacy*, Oakland, CA, pp 199-210, 1995.

[4] B. George and J. Haritsa, "Secure Transaction Processing in Firm Real-Time Database System", In *Proceedings of ACM SIGMOD*, pp462-473, 1997.

[5] J. R. Haritsa, M. J. Carey, and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control", In *11th IEEE Real-Time Systems Symposium*, 1990.

[6] J. R. Haritsa, M. J. Carey, and M. Livny, "On Being Optimistic about Real-Time", In *Proceedings of the 1990 ACM PODS Symposium*, pp331-343, April 1990.

[7] Soek-Hee Hong, Myung-Ho Kim, Yoon-Joon Lee, "Methods of Concurrency Control in Real-Time Database", In *Korean Information Science Society Review*, Vol. 11, No. 1, pp 26-36, 1993.

[8] Thomas F. Keefe, W. T. Tsai, Jaideep Srivastava, "Database Concurrency Control in Multilevel Secure Database Management Systems", In *IEEE Transaction on knowledge and Data Engineering*, vol 5, no. 6, pp1039-1055, 1993.

[9] Young-kuk Kim and Sang H. Son, "Predictability and Consistency in Real-Time Database Systems", *Advances in Real-Time Systems*, S. H. Son (ed.), Prentice Hall, pp 509-531, 1995.

[10] J. Lee and S. H. Son, "Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems", In *14th IEEE Real-Time Systems Symposium*, pp66-75, 1993.

[11] J. Mcdermott and S. Jajodia, "Orange Locking Channel-Free Database Concurrency Control via Locking", In *IFIP WG 11.3 Workshop in Database Security*, pp267-284, 1992.

[12] Ira S. Moskowitz and Myong H. Kang, "Covert Channels - Here to Stay?", In *Proceedings of COMPASS 94*, pp235-243, 1994.

[13] R. Mukkamala and S. H. Son, "A Secure Concurrency Control Protocol for Real-Time Databases", In *Annual IFIP WG 11.3 Conference of Database Security*, Rensselaerville, New York, Aug. pp 235-253, 1995.

[14] Chanjung Park, Seog Park, and Sang H. Son, "Priority-driven Secure Multi-version Locking Protocol for Real-Time Secure Database Systems", In *Proceedings of IFIP 11th Working Conference on Database Security*, August 1997.

[15] Chanjung Park and Seog Park, "Alternative Correctness Criteria for Multi-version Concurrency Control and a Locking Protocol via Freezing", In *International Database Engineering and Applications Symposium (IDEA '97)*, pp. 73-81, August 1997.

[16] S. H. Son and R. David, "Design and Analysis of a Secure Two-Phase Locking Protocol," In *18th International Computer Software and Applications Conference (COMPSAC'94)*, Taipei, Taiwan, pp 374-379, 1994.

[17] S. H. Son, R. David, and B. Thuraisingham, "An Adaptive Policy for Improved Timeliness in Secure Database Systems", In *Annual IFIP WG 11.3 Conference of Database Security*, pp223-233, 1995.

Array Reference Allocation Using SSA-Form and Live Range Growth

Marcelo Cintra[1] and Guido Araujo[2]

[1] Conexant Systems Inc.
Newport Beach, CA, USA
marcelo.cintra@conexant.com
[2] IC-UNICAMP
Campinas, SP, Brazil
guido@ic.unicamp.br

Abstract. Embedded systems executing specialized programs are playing an increasing role in computing. This trend has increased the demand for processors that can guarantee high-performance under stringent cost, power and code size constraints. Indirect addressing is by far the most used addressing mode in programs running on these systems, since it enables the design of small and faster instructions. This paper proposes a solution to the problem of allocating registers to array references using auto-increment addressing modes. It extends previous work in the area, by enabling efficient allocation in the presence of control-flow statements. An optimizing DSP compiler, from Conexant Systems Inc., was used to validate this idea. Experimental results reveal a substantial improvement in code performance, when comparing to a priority-based register coloring algorithm.

1 Introduction

Embedded programs, like those used in audio/video processing and telecommunications, are playing a crescent role in computing. Due to its performance and code size constraints, many embedded programs are written in assembly, and run in specialized processors and/or commercial CISC machines. The increase in the size of embedded applications has put a lot of pressure towards the development of optimizing compilers for these architectures. Processors that run embedded programs range from commercial CISC machines (e.g. Motorola 68000) to specialized *Digital Signal Processors* (DSPs) (e.g. ADSP 21000 [2]), and encompass a considerable share of the processors produced every year.

Address computation takes a large fraction of the execution time for most programs. Addressing can account for over 50% of all program bits and 1 out of every 6 instructions for a typical general-purpose program [16]. In order to speed-up address computation, most embedded processors offer specialized *addressing modes*. A typical example is the auto-increment (decrement) mode [1],

[1] In this paper we use post-increment only. A generalization to include pre-increment is fairly straightforward.

J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 48–62, 2001.

which enables the encoding of very short instructions. All commercial DSPs and most CISC processors *Instruction Set Architectures* (ISAs) have auto-increment (decrement) modes. Although register increment operations can be accommodated in the slot of a VLIW instruction, modern VLIW machines, like the TMS320C6x [23], also have auto-increment modes, given that it saves one instruction slot which can be used to compact other operations.

Global register allocation is an important problem in code generation which has been extensively studied [8,6,9,15]. Other researchers have considered the interaction of register allocation and scheduling in code generation for RISC machines [14,5], and interprocedural register allocation [7]. Horwitz et al [17] proposed the first algorithm for optimal allocation of address registers in straight line code. The allocation of local variables to the stack-frame, using auto-increment (decrement) mode, has been studied in [4,20,22,18,12].

```
(1)   for (i = 1; i < N-1; i++) {
(2)     avg = a[i] >> 2;
(3)     if (i % 2) {
(4)       avg += a[i-1] << 2;
(5)       a[i] = avg * 3;
(6)     }
(7)     if (avg < error)
(8)       avg -= a[i+1] - error/2;
(9)
(10)  }
(11)
```

(a)

```
p = &a[1];
for (i = 1; i < N-1; i++) {
  avg = *p++ >> 2;
  if (i % 2) {
    p += -2;
    avg += *p++ << 2;
    *p++ = avg * 3;
  }
  if (avg < error)
    avg -= *p - error/2;
}
```

(b)

Fig. 1. (a) Code fragment; (b) Modified code that enables the allocation of one register to all references.

The *Array Reference Allocation* (ARA) is the problem of allocating address registers to array references in order to minimize the number of registers and instructions required to update them. ARA has been studied before in [3,13, 19]. These are efficient graph-based solutions to the problem, when references are restricted to basic block boundaries. In this paper, we propose a global reference allocation approach that extends previous work beyond basic block boundaries. A real-world optimizing DSP compiler from Conexant Systems Inc. is used to validate the technique. This paper is divided as follows. Section 2 describes the array reference allocation problem. Section 3 proposes a solution to this problem. Finally, Section 4 evaluates the performance of the algorithm, and Section 5 summarizes the main results of this work.

2 Array Reference Allocation

Array Reference Allocation is the problem of allocating address registers (`ar's`) to array references such that the number of simultaneously live address registers is kept below the maximum number of address registers in the processor, and the number of new instructions required to do that is minimized. In many compilers, this is done by assigning address registers to similar references in the loop. This usually results in very efficient code, when traditional optimizations like induction variable elimination and loop invariant removal [1,21] are in place. Nevertheless, assigning the same address register to different instances of the same reference does not always result in the best code. For example, consider the code fragment of Figure 1(a). If a register is assigned to each distinct reference $a[i-1], a[i], a[i+1]$, three address registers are required for the loop. Now, assume that only one address register is available in the processor. If we rewrite the code from Figure 1(a) using a single pointer `p`, as shown in Figure 1(b), the resulting loop uses only one address register that is allocated to `p`. The cost of this approach is the cost of a pointer *update instruction* (`p += -2`) introduced in one of the loop control paths, which is considerably smaller than the cost of spilling two registers. The C code fragment of Figure 1(b) is a source level model of the *intermediate representation* code resulting after we apply the optimization described in the next sections.

3 Live Range Growth

Assume that loop induction variables are updated using *affine* functions, that the order of the array references, with respect to each other, is known and that the array data type is a memory word (a typical characteristic of embedded programs). Moreover, references are kept atomic in the compiler intermediate representation, and array subscript expressions are not considered for *Common Sub-expression Elimination* (CSE). In our notation, little squares represent array references, and squares with the same color define a live range (e.g. ranges R and S in Figure 4(a)-(e)). All references in a live range share the same address register. An edge between two references of a live range indicates that the references are glued together through auto-increment mode or an update instruction.

To solve the Array Reference Allocation problem we developed a technique that repeatedly *merges* pairs of live ranges R and S, such that all references in the new range (called $R \bowtie S$) are allocated to the same register. The algorithm starts by assigning a range to each reference in the loop. Ranges are merged pairwise until its number is smaller or equal to the number of address registers available in the processor. We call this approach *Live Range Growth* (LRG). The cost of merging two ranges is the total number of cycles of the update instructions required by the merge. Inner loops are treated first, hierarchically followed by outer loops. The algorithm is greedy and its runtime complexity is $O(n^2)$, where n is the number of references in the loop. Despite its complexity, the experimental results (Section 4) reveal that LRG is fast (no substantial increase in runtime was detected) and considerably improves address register allocation.

3.1 The Indexing Distance

As discussed above, in order to bound allocation to the number of address registers in the processor, sometimes it is desirable that two references share the same register. This is done by inserting an instruction between the references or by using the auto-increment (decrement) mode. The possibility of using auto-increment (decrement) can be measured by the *indexing distance*.

Definition 1. *Let a and b be array references and s the increment of the loop containing these references. Let $index(a)$ be a function that returns the subscript expression of reference a. The indexing distance between a and b is the positive integer:*

$$d(a,b) = \begin{cases} |index(b) - index(a)| & \text{if } a < b \\ |index(b) - index(a) + s| & \text{if } a > b. \end{cases} \quad (1)$$

where $a < b$ ($a > b$) if a (b) precedes b (a) in the schedule order.

Consider, for example, array references from lines (8) and (2) of Figure 1(a) (i.e. $a[i+1]$ and $a[i]$), where $s = 1$. The indexing distance $d(a[i+1], a[i]) = |(i) - (i+1) + 1| = 0 \leq 1$. Since the indexing distance is smaller/equal to one, no update instruction is needed to update the register allocated to the address register assigned to $a[i+1]$ such that it ends up pointing to $a[i]$ in the next iteration.

The Definition 1 for indexing distance can only be applied to unidimensional arrays. When nested loops or multidimensional arrays are present, the indexing distance should take into consideration the layout of the array in memory and the size of the dimensions. Similarly as in the unidimensional case, only references for which the index function at each dimension is an affine function must be considered.

In the following, assume that the size of each array element is a memory word, and that arrays are stored in *row major*. Let j be an induction variable of a given loop L with step $s = 1$, and $r[i_1]...[i_n]$ a n-dimensional array reference in L. Each subscript index of reference r can be represented by a triple $t = (a, j, b)$, a e b are integers. Hence, reference r can be represented by a set of triples $\{(a_1, j_1, b_1), ..., (a_n, j_n, b_n)\}$.

Assume that the subscript index expression i_k is a linear function of induction variable j, i.e. $i_k = (a_k, j, b_k)$. If $k = n$, then consecutive iterations of L result in a contiguous memory access pattern, where any two consecutive memory references to elements of r are in adjacent memory positions. In this case, auto-increment mode can be used to update the address register pointing to r. On the other hand, if $k \neq 1$ then any consecutive references to r result in a sequence of accesses to memory positions that are separated from each other by an amount that depends on the size of the array dimensions that are greater than k. We call this amount the *dimensional shift* (D_k) of dimension k , and compute it (below)

```
(1)   for (i=0;i<N;i++){
(2)     for (j=0;j<N;j++){
(3)       for (k=0;k<N;k++)
(4)         c[i][j] += a[i][k]*b[k][j];
(5)     }
(6)   }
```

Fig. 2. Matrix multiplication algorithm.

as the product of the size of all dimensions of r that are greater than k.

$$D_k = \begin{cases} 1, & if\ \text{k} = \text{n} \\ \prod_{j=k+1}^{n} size(j) & \text{otherwise.} \end{cases} \tag{2}$$

where $size(j)$ is the size of dimension j of r. Based on the dimensional shift concept, we can now generalize the indexing distance for n-dimensional array references.

Definition 2. *Let r_1 and r_2 be two n-dimensional array references represented respectively by the set of triples $\{(a_1, j_1, b_1), \ldots, (a_n, j_n, b_n)\}$ and $\{(c_1, j_1, d_1), \ldots, (c_n, j_n, d_n)\}$. The indexing distance between r_1 and r_2 is the integer value:*

$$d(r_1, r_2) = \begin{cases} \sum_{k=1}^{n} (|c_k - b_k)| * D_k) & if\ r_1 < r_2 \\ \sum_{k=1}^{n} (|c_k - b_k + a_k * s)| * D_k) & if\ r_1 > r_2. \end{cases} \tag{3}$$

Example 1. Consider, for example, the algorithm of Figure 2, which computes the product of two matrices `a` and `b` and stores the result into matrix `c`. All matrices are bi-dimensional arrays were $size(d) = 50$, for any dimension $d \leq 2$.The subscript indecies of all array references in Figure 2 are linear functions of the induction variables i, j or k. Consider references `a[i][k]` and `b[k][j]` in line (4) of the inner loop. For the sake of simplicity, call these references by the name of the array they refer to, i.e. a and b. Assume that register $ar1$ ($ar2$) has been allocated to reference a (b). The indexing distance between reference a, in the current iteration, and the same reference in the next loop iteration is, according to Definition 2, $d(a, a) = 1$. Hence, auto-increment mode can be used to update register $ar1$ in the current iteration such that it points to reference a in the next iteration. On the other hand, the indexing distance between b and b in the next iteration is $d(b, b) =\mid 1 - 1 + 0 \mid * 1+ \mid 1 - 1 + 1 \mid * 50 = 50$. Since the indexing distance between two consecutive accesses to b is greater than one, auto-increment mode cannot be used for $ar2$. Therefore, an update instruction $ar2+ = 50$ is required, at the end of the inner loop, to redirect $ar2$ such that it points to $b[k][j]$ in the next iteration.

In order to enable references to share the same address registers, we have to convert them to a new representation that satisfies the following requirement:

any reference b should be reached by only one reference a, given that we want to compute $d(a, b)$ at compile time, in order to decide between using auto-increment (decrement) mode for a, or inserting an update instruction on the path from a to b. We have realized that this requirement can be satisfied if the references in the CFG are in *Static Single Assignment (SSA) Form* [10,11]. We call this variation of SSA by the name *Single Reference Form* (SRF) and describe it below.

3.2 The Single Reference Form (SRF)

Let G be some program *Control-Flow Graph* (CFG) and R the set of array references in G. G is in SRF, with respect to R, if each reference is reached by a single reference. Translating a CFG to SRF can be done using the so-called ϕ-functions [10], at the join nodes of G that are relevant to R. In order to merge ranges that converge to the same block B we first insert, at the entry of B, a dumb ϕ-function $\phi(a, a, \ldots, a)$, with the number of argument positions equals to the number of control-flow predecessors and successors[2] of B. This function is assigned to a new reference w, computed by the *ϕ-instruction* $w = \phi(a, a, \ldots, a)$. The goal of the ϕ-instructions is to sort out which references reach a given block B (see Figure 4(a)). Notice that the result of a ϕ-instruction (i.e. w) is a *virtual reference* generated by an update instruction, or by the auto-increment (decrement) mode assigned to the references that are arguments of the ϕ-function.

Similarly as in the case of SSA translation, ϕ-functions are required only at certain basic blocks. These blocks are given by the *iterated dominance frontier* [10] of the set that contains all references. Our work uses the algorithm for iterated dominance frontier described in [10]. We will not describe here any further details on how to convert a program into its SSA Form. The interested reader should report to [10].

The next step in our approach is to substitute ϕ-instructions for update instructions whenever this is required. In order to do that, we need to compute two sets: (a) the set of references that reach ϕ-instructions; and (b) the set of references that are reached by the result of ϕ-instructions. We call the technique used to compute these sets *Reference Analysis*, and based it on dataflow analysis of the program CFG [1,21].

3.3 Reference Analysis

The data item used during reference analysis is the array reference. We say that statement s *generates* reference a if a is used in s, and a is a reference from the live ranges R or S that we want to merge (i.e. $a \in R \cup S$). Statement s *kills* $a \in R \cup S$, if it uses some other reference $b \in R \cup S, b \neq a$. Based on this notation, we can determine, for each basic block B, the following dataflow sets: (a) $r\_gen[B]$, the set of references generated in B; (b) $r\_kill[B]$, the set of references killed in B. *Reference analysis* is the problem of computing the array

[2] Successors are not considered in the SSA Form.

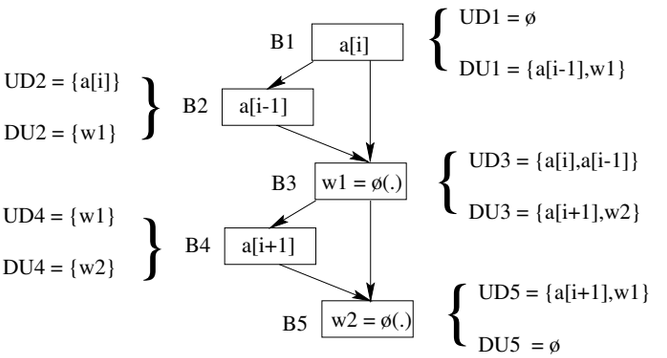


Fig. 3. CFG fragment after ϕ-instruction insertion and reference analysis.

references from $R \cup S$ that reach any given statement. It can be formulated as a *reaching definitions* problem for which there are well known solutions [1,21].

After reference analysis is performed we have, at each program point, the set of references reachable at that point. We use this to compute, for each statement s, the following sets: (a) UD_s, the *ud-chain* of the references that reach s. (b) DU_s, the set of references that use a reference defined at s. When s is a ϕ-instruction, references in UD_s are used to rename the ϕ-function arguments to $\phi(a_1, a_2, \ldots, a_n, b_1, \ldots, b_j, \ldots, b_m)$, $a_i \in UD_s$ and $b_j \in DU_s$. For the sake of simplicity, we represent this ϕ-function as $\phi(UD_s, DU_s)$. For example, the sets for the statement s_ϕ, in Figure 4(a), are: $UD_\phi = \{a_1, \ldots, a_i, \ldots, a_n\}$ and $DU_\phi = \{b_1, \ldots, b_j, \ldots, b_m\}$

Example 2. Consider, for example, the CFG fragment shown in Figure 3, after ϕ-instructions are inserted into blocks B_3 and B_5. The UD and DU sets for all instructions which perform array references are shown in Figure 3. We assume here that the program begins (ends) before (after) block B_3 (B_5). Notice that, if ϕ-instruction $w_1 = \phi(.)$ had not been inserted into block B_3, the instruction associated to reference $a[i+1]$ would be reached by two references, i.e. $UD_4 = \{a[i-1], a[i]\}$. In this case, it would not be possible to determine, at compile time, which of the these two references reach $a[i+1]$, and hence if auto-increment mode could be used by their instructions to point to $a[i+1]$. After instruction $w_1 = \phi(.)$ is inserted, the ud-chain at B_4 becomes $UD_4 = \{w1\}$ and thus only one reference reaches $a[i+1]$, and thus the decision can be made once the value for w_1 is computed.

3.4 Update Instruction Cost Estimate

In our approach, the pair of live ranges R and S selected for merging is the one that results in the smallest update instruction cost over all pairs of ranges available. Before merging two ranges R and S, we have to estimate the cost of

the update instructions required to do that, so as to decide if the cost of this merge is smaller than the current minimum merge cost. Consider, for example, two live ranges R and S that cross at basic block B, as shown in Figure 4(a). By definition, after the program is in SRF, each reference $b \in R \cup S$ has associated to it a set UD which contains a single reference a also from $R \cup S$. During merge, a and b are glued together by using auto-increment (decrement) mode at a or by inserting an update instruction between a and b. References that are assigned auto-increment (decrement) modes are rewritten to take this into consideration, since they will be used in future steps of the algorithm. The cost of an update instruction is measured by the number of its cycles, which we assume to be one. Let DU_a be the def-use chain of the references that are reachable from a. The cost of adding an update instruction from reference a to $b \in DU_a$ can be defined as:

$$cost(a,b) = \begin{cases} 0, & if\ |d(a,b)| \leq 1 \text{ and } a \text{ is real, or} \\ & if\ |d(a,b)| = 0 \text{ and } a \text{ is virtual,} \\ 1, & otherwise. \end{cases} \quad (4)$$

The first condition for the zero cost, in the equation above, aims at detecting the possibility of assigning an auto-increment (decrement) to a real reference a. The second condition for the zero cost measures if the virtual reference (resulting from a ϕ-instruction) is equal to the reference b that it reaches. In all other cases, an update instruction is required and the cost is 1.

Let $Pred(B)$ ($Succ(B)$) be the set of predecessors (successors) of block B that contains ϕ-instruction s_ϕ, and `ar` the address register assigned to $R \bowtie S$. Consider all statements s_ϕ that have at least one reference from R and S as arguments. Notice that the other arguments of ϕ are irrelevant given that we want to merge only ranges R and S. After reference analysis, any s_ϕ has associated to it sets UD_ϕ and DU_ϕ, with elements $a_i \in UD_\phi$ and $b_j \in DU_\phi$. The value of w depends on how distant the elements in UD_ϕ and DU_ϕ are. We want to select a reference w that reduces the number of update instructions required to merge all a_i to all b_j. This can be done using the following algorithm that is illustrated in Figure 4(b)-(e). In those figures, symbol "`++`" ("`--`") following a reference assigns a post-increment (decrement) mode to that reference.

1. If the elements $b \in DU_\phi$ are the same, but elements of UD_ϕ are different. In this case, w is the array reference that minimizes the update cost $\sum_{i=1}^{|UD_\phi|} cost(a_i, w) + cost(w, b)$. Remove s_ϕ and insert into B an update instruction $ar+ = d(w,b)$, if $cost(w,b) = 1$. Insert, at the exit of the block $P_i \in Pred(B)$ that contains a_i, an update instruction $ar+ = d(a_i, w)$, whenever $cost(a_i, w) = 1$; otherwise, use the adequate auto-increment (decrement) mode for a_i.

   Example 3. Consider, for example, the live ranges in Figure 4(b). Notice that $w = a[i-1]$ and all references reachable from s_ϕ are the same, i.e. $b_j = a[i+2], \forall b_j \in DU_\phi$. Since $cost(a[i-1], a[i+2]) = 1$ we substitute s_ϕ

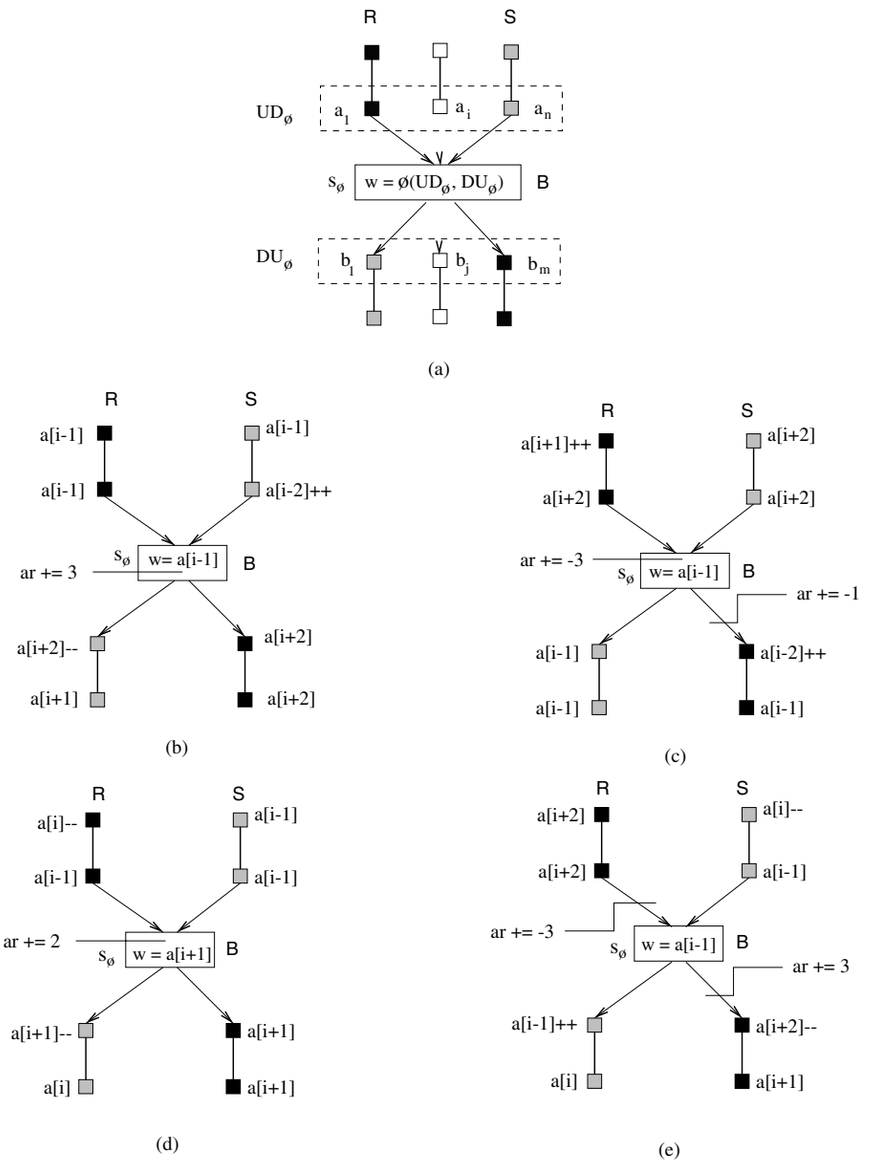


Fig. 4. Examples of update instruction insertion.

by an instruction $ar+=3$ that redirects ar from $w=a[i-1]$ to $a[i+2]$. Reference $a[i-2]$ is assigned auto-increment mode, and thus it is rewritten to $a[i-1]$ for the future steps of the algorithm.

2. If the elements $a \in UD_\phi$ are the same, but elements of DU_ϕ are different.
In this case w is the array reference that minimizes the cost $cost(a,w) + \sum_{j=1}^{|DU_\phi|} cost(w,b_j)$. Remove s_ϕ and insert into B update instruction $ar+=d(a_i,w)$, if $cost(a_i,w)=1$, otherwise use the adequate auto-increment (decrement) mode for a_i. Insert, at the entry of block $S_j \in Succ(B)$ that contains b_j an update instruction $ar+=d(w,b_j)$, if $cost(w,b_j)=1$.

   Example 4. Consider, for example, the live ranges in Figure 4(c). Here $w=a[i-1]$ and all references that reach s_ϕ are the same, i.e. $a_i=a[i+2], \forall a_i \in UD_\phi$. Since $cost(a[i+2],a[i-1])=1$, we substitute s_ϕ by an instruction $ar+=-3$ that redirects ar from $a[i+2]$ to $w=a[i-1]$. Moreover, since $cost(a[i-1],a[i-2])=1$, we insert $ar+=-1$ on the path from $w=a[i-1]$ to $a[i-2]$.

3. If the elements of DU_ϕ (UD_ϕ) are the same.
In this case, w is the reference that minimizes $cost(a,w)+cost(w,b)$. Remove s_ϕ and insert update instructions or use auto-increment (decrement) mode according to steps (1) and (2) above.

   Example 5. Consider, for example, the live ranges in Figure 4(d). Notice that $w=a[i+1]$ and all references in UD_ϕ (DU_ϕ) are the same, i.e. $a[i-1]$ (a[i+1]). Since $cost(a[i+1],a[i+1])=0$ no update instructions are required from $w=a[i+1]$ to $a[i+1]$. On the other hand, $cost(a[i-1],a[i+1])=1$ and thus an instruction $ar+=2$ is required in block B.

4. If the elements in UD_ϕ (DU_ϕ) are not the same.
In this case, w is the array reference that minimizes the update cost $\sum_{i=1}^{|UD_\phi|} cost(a_i,w) + \sum_{j=1}^{|DU_\phi|} cost(w,b_j)$. Remove s_ϕ and insert, at the exit of the block $P_i \in Pred(B)$ that contains a_i, an update instruction $ar+=d(a_i,w)$, if $cost(a_i,w)=1$; otherwise, use auto-increment (decrement) mode for a_i. Insert an update instruction $ar+=d(w,b_j)$ at the entry of $S_j \in Succ(B)$, if $cost(w,b_j)=1$.

   Example 6. Consider, for example, the ranges in Figure 4(e). In this case $w=a[i-1]$, and the elements of sets UD_ϕ and DU_ϕ are distinct within each set. Since $cost(a[i+2],a[i-1])=1$ we insert at the end of the block that contains $a[i+2]$ an update instruction $ar+=-3$ to redirect ar from $a[i+2]$ to $w=a[i-1]$. Similarly, instruction $ar+=3$ is inserted at the entry of the block that contains $a[i+2]$.

3.5 The LRG Algorithm

After a program is in SRF, any loop basic block to which array references converge will have a ϕ-instruction, including the header and the tail of the loop.

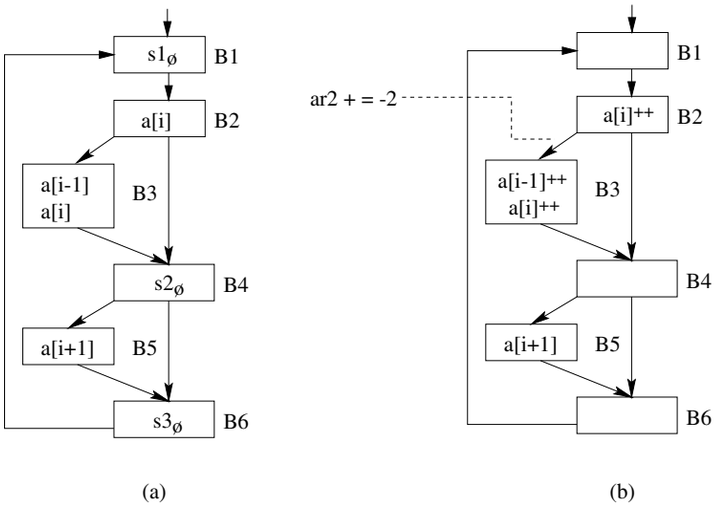


Fig. 5. The case of the fragment in Figure 1:(a) ϕ-function insertion; (b) Inserting auto-increment mode and update instructions.

As a consequence, the set of ϕ-instructions in the loop blocks forms a system of assignments which evaluates according to the rules discussed in Section 3.4. The unknown variables in this system are the results w_k of the ϕ-functions. These equations have circular dependencies, caused basically by the following reasons: (a) it originates from the loop CFG; (b) each ϕ-function depends on its sets UD_ϕ and DU_ϕ. Therefore, any optimal solution for variables w_k cannot always be determined exactly. Consider, for example, the CFG of Figure 5(a) corresponding to the loop in Figure 1, after ϕ-instructions are inserted and reference analysis is performed. Three ϕ-function assignments result in blocks B_1, B_4 and B_6, namely:

$$s1_\phi : w_1 = \phi(w_3, a[i])$$
$$s2_\phi : w_2 = \phi(a[i], a[i], a[i+1], w_3)$$
$$s3_\phi : w_3 = \phi(a[i+1], w_2, w_1)$$

These assignments create circular dependencies, and thus cannot be solved exactly. Therefore, estimates for the exact values of w_k must be computed. One way to do that is to choose an evaluation order for the assignments such that, at each assignment $w_k = \phi(.)$, any argument of ϕ that is a virtual reference $w_j, j = 1, 2, 3$ is simply ignored. For example, during the estimate of value w_3 in $s3_\phi$ arguments w_1 and w_2 are ignored and the resulting assignment becomes $w_3 = \phi(a[i+1])$, which results in $w_3 = a[i+1]$. In our approach, the first assignment in the evaluation order is the one at the tail of the loop. From this point on, our algorithm proceeds backward from the tail of the loop up to the header, evaluating each assignment as they are encountered. At each step, the

rules from Section 3.4 are used to compute the new value of w_k, which is then used in the evaluation of future assignments.

Notice that any reference that is assigned an addressing mode, by one of the cases (1)-(4) in Section 3.4, should be rewritten in order to take the new mode into consideration during future steps of the algorithm. For example, after case (1) is finished in Example 3, an auto-increment mode is assigned to reference $a[i-2]$. Hence, reference $a[i-2]$ is rewritten to $a[i-1]$, which is the value used in the next steps of the algorithm. After all ϕ-instructions are evaluated, the algorithm determines the set of real references a and their corresponding sets $|DU_a|$. For each element $b_j \in DU_a$, it uses the rules from Section 3.4 to insert an update instruction from a to b_j, whenever $d(a, b_j) > 1$.

Example 7. Figure 5 demonstrates the application of the LRG algorithm to the code fragment in Figure 1(a). First, the ϕ-instruction $s3_\phi$ at the tail of the loop (block B_6) evaluates $w_3 = a[i+1]$. This reference is equivalent to $a[i]$ in the next loop iteration. In the next step, ϕ-instruction $s2_\phi$ in B_4 is evaluated using the case (3) of Section 3.4. The ϕ-function results in $w_2 = a[i+1]$ and auto-increment modes are assigned to references $a[i]$ in blocks B_2 and B_3. These references now become $a[i+1]$. Next, ϕ-instruction $s1_\phi$, in block B_1, is evaluated resulting in $w_1 = a[i]$. At this point, all virtual references (i.e. the values of w_k) have been computed. In the next step, update instructions are inserted for any real reference that still requires it. In the example of Figure 5, real reference $a[i+1]$ in block B_2 reaches reference $a[i-1]$ in block B_3. Since $d(a[i+1], a[i-1]) = 2$, then instruction $ar+=2$ is inserted on the path from B_2 to B_3. The output of this optimization is an intermediate representation code equivalent to the source-level code of Figure 1(b), which uses a single address register and just one update instruction.

4 Experimental Results

Table 1 shows the result of applying LRG to a set of typical signal processing programs, most of them from the DSPStone Benchmark. Unfortunately, due to its specialized and proprietary nature, not many large DSP programs are available to the public. On the other hand, those available form the core of most DSP applications, and correspond to the majority of their execution time, what still makes them useful for DSP benchmarking. We have implemented our optimization in an optimizing DSP compiler from Conexant Systems Inc., which uses an efficient priority-based register coloring allocation algorithm. This is a production quality optimizing compiler that implements all relevant machine optimizations described in [1], and that is used to compile real-world applications. The experimental results show that LRG results on an average 11% speed-up, when comparing with the combination of the original allocation algorithm with induction variable elimination, loop invariant removal, and other traditional optimizations that support address register allocation. The average size overhead due to update instructions was 0.65%.

Table 1. Live Range Growth speed-up and size overhead.

Program Name	Priority-based		LRG Optimized		Comparison (%)	
	Cycles	Size	Cycles	Size	Speedup	Size
convenc	4331	4667	3943	4647	9%	0%
convolution	1220	2068	1042	2077	17%	+1%
dot_product	165	1305	160	1269	3%	-2%
biquad_N_sections	1380	2980	1218	2905	13%	-2%
fir_array	1471	2626	1263	2666	16%	+2%
fir2dim	7684	4546	6728	4566	14%	+1%
lms_array	2276	3644	1919	3665	18%	+1%
mat1x3	1202	2668	1113	2705	7%	+2%
matrix1	34657	3057	30520	3135	13%	+3%
n_complex_updates	2985	3300	2336	3410	27%	+4%
n_real_updates	1855	2716	1452	2785	27%	+3%
fft	173931	10103	165549	10097	5%	0%
autcor	179633	4003	167238	3990	7%	0%
fir8	280324	5143	256476	5088	9%	-1%
latsynth	3115	3408	3050	3402	2%	0%
fir_lms2	3454	3353	3317	3298	4%	-1%
latanal	703662	3425	691662	3411	2%	0%

5 Conclusions

This paper proposes a technique that improves address register allocation for auto-increment (decrement) addressing modes. It uses SSA form and a simple, and yet effective algorithm, to extend previous work in the area towards considering allocation beyond basic blocks. Experimental results reveal an average 11% performance improvement when comparing to a priority-based register coloring technique.

6 Acknowledgments

The authors thank Alan Taylor, Keith Bindloss and the compiler group at Conexant Systems Inc. for their strong support to this project. This work was partially supported by CNPq (300156/97-9), ProTem CNPq/NSF Collaborative Research Project (68.0059/99-7) and FAPESP (98/06225-2-R). We also thank the reviewers for their comments.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Addison Wesley, Boston, 1988.
2. Analog Devices. *ADSP-2100 Family User's Manual*.
3. G. Araujo, A. Sudarsanam, and M. S. Instruction set design and optimizations for address computation in DSP processors. In *9th International Symposium on Systems Synthesis*, pages 31–37. IEEE, November 1996.
4. D. H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Software Practice and Experience*, 22(2):101, February 1992.
5. D. Bradlee, E. S.J., and R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, April 1991.
6. P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proc. of the ACM SIGPLAN'89 on Conference on Programming Language Design and Implementation*, pages 98–105, June 1982.
7. D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proc. of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 192–202, June 1991.
8. G. Chaitin. Register allocation and spilling via graph coloring. In *Proc. of the ACM SIGPLAN'82 Symposium on Compiler Construction*, pages 98–105, June 1982.
9. F. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, October 1990.
10. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form. In *Proc. of the ACM POPL'89*, pages 23–25, 1989.
11. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the program dependence graph. *ACM TOPLAS*, 13(4):451–490, October 1991.
12. E. Eckstein and A. Krall. Minimizing cost of local variables access for DSP-processors. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 20–27, May 1999.
13. C. Gebotys. DSP address optimization using a minimum cost circulation technique. In *Proceedings of the International Conference on Computer-Aided Design*, pages 100–103. IEEE, November 1997.
14. J. Goodman and A. Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 1988 Conference on Supercomputing*, pages 442–452, July 1988.
15. R. Gupta, M. Soffa, and D. Ombres. Efficient register allocation via coloring using clique separators. *ACM Trans. Programming Language and Systems*, 16(3):370–386, May 1994.
16. Hitchcock III, C.Y. *Addressing Modes for Fast and Optimal Code Generation*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, Dec. 1986.
17. L. Horwitz, R. Karp, R. Miller, and S. Winograd. Index register allocation. *Journal of the ACM*, 13(1):43–61, January 1966.
18. R. Laupers and F. David. A uniform optimization technique for offset assignment problems. In *Proceedings of the ACM SIGDA 11th International Symposium on System Synthesis*, pages 3–8, December 1998.

19. R. Leupers, A. Basu, and P. Marwedel. Optimized array index computation in DSP programs. In *Proceedings of the ASP-DAC*. IEEE, February 1998.
20. S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. In *Proc. of 1995 ACM Conference on Programming Language Design and Implementation*, 1995.
21. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
22. A. Rao and S. Pande. Storage assignment optimizations to generate compact and efficient code on embedded dsps. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 128–138, May 1999.
23. Texas Instruments. *TMS320C6x User's Guide*, 1999.

PROPAN: A Retargetable System for Postpass Optimisations and Analyses

Daniel Kästner*

Saarland University
Fachbereich 6.2 Informatik
Saarbrücken, Germany
kaestner@cs.uni-sb.de

Abstract. PROPAN is a system that allows for the generation of machine-dependent postpass optimisations and analyses on assembly level. It has been especially designed to perform high-quality optimisations for irregular architectures. All information about the target architecture is specified in the machine description language TDL. For each target architecture a phase-coupled code optimiser is generated which can perform integrated global instruction scheduling, register reassignment, and resource allocation by integer linear programming (ILP). All relevant hardware characteristics of the target processor are precisely incorporated in the generated integer linear programs. Two different ILP models are available so that the most appropriate modelling can be selected individually for each target architecture. The integer linear programs can be solved either exactly or by the use of ILP-based approximations. This allows for high quality solutions to be calculated in acceptable time. A set of practical experiments shows the feasibility of this approach.

1 Introduction

During the last years, the markets for telecommunication, embedded systems, and multimedia applications have been rapidly growing. The stringent time constraints imposed by many of those applications in connection with severe cost constraints have led to the development of specialised, irregular architectures designed to efficiently execute typical applications of digital signal processing. Such hardware features include heterogeneous register files, simultaneous execution of multiple operations, support for low-overhead looping, and restricted connectivity of functional units to register files or buses.

In the area of general-purpose processors, compiler technology has reached a high level of maturity. However, the code quality achieved by traditional high-level language compilers for irregular architectures often cannot satisfy the requirements of the target applications [31,37]. Generating efficient code for irregular architectures requires highly optimising techniques that must be aware of

* Member of the Graduiertenkolleg "Effizienz und Komplexität von Algorithmen und Rechenanlagen" (supported by the DFG).

J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 63–80, 2001.

specific hardware features of the target processor. Since such techniques are usually not employed in standard compilers, many DSP applications are developed in assembly language. Due to the increasing complexity of typical applications and the shrinking design cycles of embedded processors, this approach becomes increasingly unacceptable. As a solution this article proposes retargetable postpass optimisation techniques that can be quickly adapted to different target architectures and yet produce high quality code. The goal of our work is the development of a fully retargetable, phase-coupled postpass optimisation framework that allows for an easy integration of program analyses and user-supplied optimisation routines.

1.1 The Role of Postpass Strategies

Although several retargetable research compilers have been developed during the last years (see Sec. 2), the use of such systems in industry is still rare. One reason is that changing the compiler leads to very high costs. So the question can be asked whether it is possible to improve the quality of existing compilers that fail to produce satisfying code for irregular architectures. A promising approach is using a postpass framework that can process inputs resulting from previous code generation passes. Previous studies [18] have shown that the integration of a postpass approach into existing tool chains can be done with moderate effort.

In embedded systems, the software often has to meet specific requirements that necessitate complex program analyses that are necessarily postpass analyses. An example is the static cache behaviour prediction of [7]. The results of such analyses can also be used for program optimisations, e. g. by cache-sensitive task scheduling algorithms [19].

1.2 The Phase Coupling Problem

The process of generating code for high level language programs can be divided into several phases: code selection, register allocation, instruction scheduling, register assignment, and resource allocation. The goal of instruction scheduling is to rearrange a code sequence in order to exploit instruction level parallelism. The task of register allocation is to decide which variables and expressions of the intermediate representation are mapped to registers and which ones are kept in memory. The goal is to minimise the number of memory references during program execution. Register assignment is the task of determining the physical register that is used to store a value that has been previously selected to reside in a register[1]. The task of resource allocation is to allocate resources, e. g. functional units or buses to operations (see Sec. 5).

Most of these tasks are $\mathcal{NP}$-hard problems. In classical approaches they are solved separately by heuristic methods. However all these phases are interdependent, i. e. , decisions made by one phase impose constraints on the following

[1] Often the term register allocation is used in a more general sense to denote both the phases of register allocation proper and register assignment.

phases possibly leading to the generation of inefficient code. That problem is known as the phase coupling problem. In order to generate high-quality code for irregular architectures, it is essential that the phase coupling is addressed. However, due to complexity reasons a full phase integration is usually not possible. Instead, a promising approach is to identify several phase groupings such that each of those groups is solved integratedly. One reasonable partition is a coupling of instruction selection and register allocation on the one hand, and of instruction scheduling, register assignment, and resource allocation on the other hand.

1.3 Overview of the Paper

After an overview of related work in Sec. 2, the PROPAN framework is introduced in detail in Sec. 3. First, a mechanism of specifying the target architecture is presented. Then, Sec. 3.2 gives a detailed overview of the structure of PROPAN. In Sec. 4, the basic ILP models are introduced and a summary of the implemented model extensions is given. Those extensions concern the modelling of the control flow structure of the program and the incorporation of user-specified logical constraints. Section 5 gives an overview of experimental results; Sec. 6 summarises the contributions of our work and gives an outlook.

2 Related Work

In this overview, we will concentrate on code generation systems for irregular architectures. There are several approaches where phase coupling is done by heuristic methods. CHESS [25] has been designed as a retargetable code generation framework for fixed point DSPs. The target processor is modelled by instruction set graphs that are automatically extracted from an nML-description of the processor [6]. It is assumed that each instruction is executed in one machine cycle. Code selection, register allocation, and instruction scheduling are implemented as separated modules. The register allocation decisions are guided by estimates about the effects on instruction scheduling. This way, information is exchanged between the phases, but there is no true phase integration.

The retargetable CBC compiler [25] also specifies the target architecture using the nML language. An extended list scheduling algorithm is used that addresses the problems of register allocation and instruction scheduling in an integrated way. The scheduling and allocation decisions are guided by heuristics.

Express [13] integrates code selection and register allocation into instruction scheduling in a heuristic way (mutation scheduling) [26]. The information about the hardware is extracted from the machine description language Expression [13].

Apart from the heuristic approaches, there are also phase-integrated methods that allow to calculate exact, optimal solutions — usually at the cost of higher calculation times. Since in digital signal or real-time applications code quality is extremely important, higher calculation times are acceptable within reasonable

limits. The AVIV retargetable code generator [14] builds on the SPAM library which is a retargetable code generation framework for digital signal processors [32]. AVIV uses a branch-and-bound algorithm that performs functional unit assignment, operation grouping, register bank allocation, and scheduling. Detailed register allocation is carried out as a second step.

Bashford and Leupers [3] are developing a framework for phase-coupled code selection and register allocation using constraint logic programming. Alternative solutions are kept as long as possible, so that the decision among them can be delayed until the scheduling phase.

There have been only few approaches to incorporate ILP-based methods into the code generation process of a compiler. An approach for ILP-based instruction scheduling for vector processors has been presented in [2]. Wilson et al. [25] use an ILP-formulation for simultaneously performing code selection, scheduling, register allocation and assignment; movements of operations across basic block boundaries are not addressed. The complexity of the resulting formulations however leads to high computation times. Leupers has developed a retargetable compiler for digital signal processors [23] where local instruction scheduling is performed by integer linear programming. Other ILP-based approaches have been developed in the context of software pipelining, e. g. [30,11]; however those approaches mostly deal with homogeneous VLIW-like architectures.

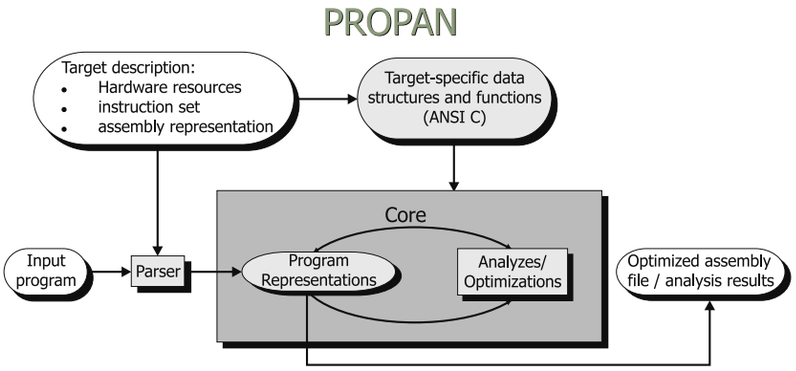


Fig. 1. The PROPAN System

3 The PROPAN Framework

In the following, the PROPAN framework is presented in more detail. PROPAN (Postpass Retargetable Optimiser and Analyser) has been designed as a retargetable framework for postpass optimisations and analyses. In Sec. 3.1, the architecture specification language TDL is presented; Sec. 3.2 gives a detailed overview of the design of PROPAN.

3.1 Specifying the Architecture

In order to allow for easy retargetability, a specification mechanism for the target architecture and its instruction set is required. That specification has to cover all information needed by optimisations and analyses and must allow a representation where the information can easily be accessed by target applications. Possible input formats are assembly code or compiler-specific intermediate code. PROPAN automatically generates a parser for the input language from the machine specification. Different analyses may require different views of the architecture. Therefore, the specification language should support different levels of abstraction and represent the information in a way that they can be easily accessed by target applications. In order to achieve high code quality, the generated optimisers must take the hardware characteristics of the target architecture precisely into account. Therefore irregular hardware properties have to be specified in a way that allows to automatically generate appropriate hardware-sensitive ILP constraints. Finally, the specification language has to be extendable and flexible in use, so that a wide range of architectures can be modelled.

There is a large number of existing architecture description formalisms, e. g. , [24], [27], [6], [28], [12], [13], [4]. Yet satisfying the demands mentioned above required the development of a dedicated specification language, called TDL (*Target Description Language*). Details about the design of TDL and a comparison to other architecture description languages can be found in [16]; in the following, only a short summary is given.

A TDL description has a modular structure. It is composed of a specification of the hardware resources, a description of the instruction set, a constraint section, an assembly section, and a pipeline section. In the resource section, all hardware resources that can influence the analyses and optimisations are introduced and their properties are specified by an extendable attribute mechanism. The description of the instruction set is given in the form of an attribute grammar [34]. The attribute mechanism allows to identify important properties of operations. In the constraint section, a set of constraints can be specified that have to be respected during code transformations in order to preserve the program semantics. This includes, e. g. , restrictions of the available parallelism of the target machine or dependences between instruction scheduling and register assignment (see Sec. 4.4). The assembly section deals with syntactic details of the assembly language. Typically the assembly language is not a mere representation of the instruction set of the processor. Assembly expressions, directives as, e. g. , data declarations or segment directives, macros, comments and different forms of operation or instruction delimiters must be known.

Each TDL description is checked for semantical consistency, so input errors and inconsistencies in the hardware description can be detected early.

3.2 The Design of PROPAN

The input of the PROPAN framework (see Fig. 1) consists of a TDL description of the target machine and of the assembly programs that are to be analysed or optimised. The TDL file has to be processed once for each target architecture and is

used in two ways: first a parser for the specified input language is automatically generated. The parser reads the input programs and calculates the control flow graph by jump target and loop detection algorithms. The control flow graph is represented in a generic format, called *CRL* (Control Flow Representation Language) [22]. This representation serves as an interface that allows additional program analyses and optimisations to be easily incorporated into the system. Second, a set of ANSI C files containing fundamental data structures and functions is generated. All relevant information about the hardware resources and the instruction set is represented by the generated data structures; the functions are provided as an interface to initialise, access and manipulate the contained information. The corresponding files can be included in any target application and allow for generic accesses to architecture-specific properties. From the control flow graph, the required program representations as, e. g. , the data dependence and the control dependence graphs are calculated. If register reassignment is to be performed, a register renaming step is executed in order to remove spurious data dependences limiting the available parallelism.

The core of the optimisations is a generic algorithm for integrated instruction scheduling, register reassignment and functional unit allocation based on integer linear programming [18]. For each input program, integer linear programs (ILP) are generated that model the execution of the program for the specified target architecture. Two alternative ILP models are available (see Sec.4), so that the most appropriate modelling can be selected individually for each target architecture. Since the calculation time for the integer linear programs may be high, they can be either solved exactly or approximatively. The basic idea of the ILP-based approximations is the iterative solution of heuristically determined relaxations of the original problem. By the use of such approximations, the calculation time is usually significantly reduced and still high-quality solutions are obtained. More details about the approximation techniques and their use in practise can be found in [18,20].

In contrast to most previous phase-integrated approaches, the optimisation scope is not restricted to a single basic block. Several basic blocks can be grouped to form a superblock, that can be extended across loop boundaries as long as the control flow paths are not disjoint [20]. The control flow graph of the input program to be optimised is covered by superblocks; the maximal size of the superblocks can be defined by the user in order to influence the computation time. For each superblock, an individual integer linear program is generated and optimised; the optimisation starts with the most frequently executed superblocks, similar to the trace scheduling mechanism [9]. In order to reduce calculation time, the ILP-based optimisations can be restricted to important code sequences that are often executed, e. g. to nested loops. Details about the algorithms that are used can be found in [20].

4 The ILP Models

ILP-formulations can be classified as order-based or time-based models [21]. In time-based formulations, the choice of the decision variables is based on the point of time the modelled event is assigned to. In order-based approaches, the decision variables reflect the ordering of the modelled events; the assignment to points of time is implicitly given.

In the area of architectural synthesis, several ILP-formulations have been developed for the problem of instruction scheduling and resource allocation [10, 36]. In [17,18] we have investigated the applicability of those approaches to the code generation problem. Based on the results of those studies, we have developed extensions of the order-based ILP model SILP [36] and of the time-based formulation OASIC [10]. The SILP model can be used to perform integrated instruction scheduling, resource allocation and register reassignment. The OASIC formulation is an alternative for processors where the register reassignment can be neglected; here the tasks of instruction scheduling and resource allocation can be integrated. As shown in [17], the integration of the register assignment phase in this model, although possible, is not practicable for complexity reasons [17].

In the following, some basic definitions are presented and then the structure of the ILP formulations is summarised and evaluated.

4.1 Basic Definitions

The task of instruction scheduling is to rearrange a code sequence in order to minimise the required execution time. During reordering, the data dependences of the operations have to be respected. The data dependences are modelled by the data dependence graph $G_D = (V_D, E_D)$ whose nodes correspond to the operations of the input program and whose edges reflect the dependences between the adjacent nodes. There are three different types of data dependences: *true dependences* (read after write, E_D^t), *output dependences* (write after write, E_D^o), *anti dependences* (write after read, E_D^a).

In order to describe the mapping of operations to resource types, the *resource graph* G_R is used that is derived from the machine description. G_R is a bipartite directed graph $G_R = (V_R, E_R)$, where $(j, k) \in E_R$ means that instruction j can be executed by the resources of type k.

4.2 An Order-Based Formulation

The SILP model is an order-based formulation. In this model, the main decision variables describe the flow of resources through the instructions of the program: $x_{ij}^k \in \{0, 1\}$ indicates whether operation i passes an instance of resource type k to operation j. The starting time for the execution of each operation is implicitly determined by the calculated resource flow.

Apart from the flow variables x_{ij}^k, the following types of decision variables are used: $t_i \in \mathbb{Z}$ denotes the starting time for the execution of operation i, $w_j \in \mathbb{Z}$ the number of control steps required to execute operation i. The latency of the

functional unit executing operation j, i.e. the minimal time interval between successive data inputs to this functional unit is represented by $z_j \in \mathbb{Z}$. Finally $R_k \in \mathbb{Z}$ denotes the number of instances of resource type $k \in V_K$.

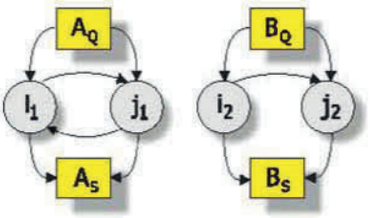


Fig. 2. Example of a Resource Flow Graph

The ILP is generated from a *resource flow graph* $G_F = (V_F, E_F)$. This graph describes the execution of a program as a flow of the available hardware resources through the operations of the program; for each resource type, this leads to a separated flow network. Each resource type $k \in V_K$ is represented by two nodes $k_Q, k_S \in V_F$ where the nodes k_Q represent the sources and the nodes k_S the sinks. The first operation to be executed on resource type k gets an instance k_r of this type from the source node k_Q; after completed execution, it passes k_r to the next operation using the same resource type. The last operation using a certain instance of a resource type returns it to k_S. The number of simultaneously used instances of a certain resource type must never exceed the number of available instances of this type. Each edge (i, j) of the resource flow graph $(i, j) \in E_F^k$ is mapped to a flow variable $x_{ij}^k \in \{0, 1\}$. A hardware resource of type k is moved through the edge (i, j) from node i to node j, if and only if $x_{ij}^k = 1$.

Fig. 2 shows an example resource flow graph for two resource types A and B. The operations that have to be executed by A are independent, so there is no restriction of the operation ordering. However in the subgraph of resource type B, we can see that there is only one feasible ordering of the operations what can, e. g., be caused by data dependences.

The basic idea of this modelling is that operations that are mapped to different resource types can be executed in parallel if this is not prevented by data dependences. The number of operations that can be simultaneously executed by the same resource type is given by the number of instances of this resource type. This way, the VLIW execution model is covered where an instruction can be composed of several microoperations that are executed in parallel. If, however, the target processor has no VLIW-like architecture, this can simply be modelled by declaring only one (possible virtual) execution resource [16].

The goal of the ILP-formulation is to minimise the execution time of the code sequence to be scheduled. The execution time is measured in control steps (clock cycles). So the objective function and the time constraints required to define the

total execution time M_{steps} can be given as follows:

$$\min \quad M_{steps} \tag{1}$$

$$t_j \leq M_{steps} \quad \forall\, j \in V_D \tag{2}$$

The precedence constraints are used to represent the data dependences of the input program. When operation j depends on operation i, then j may be executed only after the execution of i is finished.

$$t_j - t_i \geq w_i \quad \forall\, (i,j) \in E_D^t \tag{3}$$

$$t_j - t_i \geq w_i - w_j + 1 \quad \forall\, (i,j) \in E_D^o \tag{4}$$

$$t_i \leq t_j + w_j - 1 \quad \forall\, (i,j) \in E_D^a \tag{5}$$

The flow conservation constraints (equation (6)) assert that the value of the flow entering a node is equal to the value of the flow leaving that node. Moreover, each operation must be executed exactly once by one hardware component. This is guaranteed by equation (7). The resource constraints (8) are necessary, since the number of available instances of all resource types must not be exceeded.

$$\sum_{(i,j)\in E_F^k} x_{ij}^k - \sum_{(j,i)\in E_F^k} x_{ji}^k = 0 \;\forall\; i,j \in V_D \tag{6}$$

$$\sum_{\substack{k\in V_K:\\ (j,k)\in E_R}} \sum_{(i,j)\in E_F^k} x_{ij}^k = 1 \;\forall\; j \in V_D \tag{7}$$

$$\sum_{(k,j)\in E_F^k} x_{kj}^k \leq R_k \;\forall k \in V_K \tag{8}$$

Finally, the serial constraints are required which guarantee a correct sequencing among independent operations assigned to the same resource instance (equation (9)).

$$t_j - t_i \geq z_i + \Big(\sum_{\substack{k\in V_K:\\ (i,j)\in E_F^k}} x_{ij}^k - 1\Big)\alpha_{ij} \;\forall (i,j) \in E_F^k \tag{9}$$

When operations i and j are both assigned to the same resource type k, then j must await the execution of i, when a component of resource type k is actually moved along the edge $(i,j) \in E_F^k$, i.e., if $x_{ij}^k = 1$. Here α_{ij} is a constant that represents an upper bound of the maximal distance between i and j.

The total number of constraints is $\mathcal{O}(n^2)$, where n is the number of operations in the input program. The number of binary variables is bounded by $\mathcal{O}(n^2)$. More detailed descriptions including proofs are given in [36,17].

The Register Assignment Problem. The integer linear program presented above only covers the problem of instruction scheduling and resource allocation.

To take into account the problem of register assignment, the formulation has to be extended.

The central data structure is the register flow graph $G_F^R = (V_F^R, E_F^R)$ which is composed of the individual flow graphs for each register file of the target architecture. For each register file, a source and a sink node are introduced; the remaining nodes represent the operations of the input program that perform a write access to a register. Each marked edge $(i,j)_g \in E_F^R$ represents a possible flow of a register of register file $g \in G$ from i to j and is mapped to a binary flow variable $x_{ij}^g \in \{0,1\}$. If $x_{ij}^g = 1$, the same register is used to store the variables generated by operations i and j. Similar to Sec. 4.2, a set of constraints is required to ensure the well-definedness of the flow modelling. Additional constraints are used to model the lifetimes of the variables in the program: a register flow variable x_{ij}^g can only be set to 1, if j is scheduled at a control step where the life range of the variable defined by i has expired.

The number of additional constraints is $\mathcal{O}(m^2)$ where m is the number of operations in the input program performing write accesses to a register. The number of additional binary variables is bounded by $\mathcal{O}(m^2)$. More details including the complexity proofs can be found in [17].

4.3 A Time-Based Formulation

In [10] a time-based formulation is presented where the main decision variables describe the assignment of operations to control steps. The flow of the resources is determined implicitly by the operation sequencing. The main decision variables are called $x_{jn}^k \in \{0,1\}$, where $x_{jn}^k = 1$ means, that microoperation j is assigned to the nth control step ($n \geq 1$) in the generated schedule and is executed by an instance of resource type k. For each operation i $N(i)$ denotes the set of feasible control steps for the execution of operation i calculated by ASAP/ALAP-analysis [17]. Similarly to Sec.4.2, w_i denotes the execution time of operation i and t_i the starting time for the execution of operation i. However in this model, the t-variables are derived from the main decision variables x_{jn}^k:

$$t_i = \sum_{k:(i,k)\in E_R} \sum_{n \in N(i)} n \cdot x_{in}^k$$

In [17,20] the ILP formulation of [10] is adapted to our problem setting; its basic structures is presented in the following. The objective function and the time constraints are identical to the SILP formulation:

$$\min \quad M_{steps} \tag{10}$$

$$t_j \leq M_{steps} \quad \forall\, j \in V_D \tag{11}$$

The data dependences of the input program are represented by the precedence constraints which ensure that an operation j must not be started before operation i if there is a data dependence from i to j.

$$\sum_k \sum_{\substack{n_j \le n \\ n_j \in N(j)}} x^k_{jn_j} + \sum_k \sum_{\substack{n_i \ge n - w_i + 1 \\ n_i \in N(i)}} x^k_{in_i} \le 1 \ \forall (i,j) \in E^t_D,$$
$$n \in \{n' + w_i - 1 | n' \in N(i)\} \cap N(j)) \tag{12}$$

$$\sum_k \sum_{n_j \le n} x^k_{jn_j} + \sum_k \sum_{\substack{n_i \ge n - w_i + w_j \\ n_i \in N(i)}} x^k_{in_i} \le 1 \ \forall (i,j) \in E^o_D,$$
$$n \in \{n' + w_i - w_j | n' \in N(i)\} \cap N(j) \tag{13}$$

$$\sum_k \sum_{\substack{n_j \le n - w_j + 1 \\ n_j \in N(j)}} x^k_{jn_j} + \sum_k \sum_{\substack{n_i > n \\ n_i \in N(i)}} x^k_{in_i} \le 1 \ \forall (i,j) \in E^a_D,$$
$$n \in \{n' + w_j - 1 | n' \in N(j)\} \cap N(i)) \tag{14}$$

Additionally, the assignment constraints are required which ensure that the execution of an operation starts in exactly one control step and is performed by exactly one resource type.

$$\sum_k \sum_{n \in N(j)} x^k_{jn} = 1 \qquad \forall \ j \in V_D \tag{15}$$

Finally, the resource constraints guarantee that the number of available instances of each resource type must not be exceeded, so that in no control step, more than R_k operations may be executed by resource type k. Let $N_{jm} = \{n : m = n + p, \ 0 \le p \le w_j - 1\}$, then the constraints read as follows:

$$\sum_{j \in V_D} \sum_{n \in N_{jm} \cap N(j)} x^k_{jn} \le R_k \qquad \forall m \le M_{steps} \tag{16}$$

4.4 Modelling Extensions

Most architectures do not have a fully orthogonal instruction set. The digital signal processor ADSP-2106X[1] provides only a restricted parallelism between ALU and multiplier. Some operations can only be executed in parallel if all operands reside in uniquely defined register groups within the (heterogeneous) register file. If the operands are located in other registers, the operations cannot be grouped into one instruction [18]. Such restrictions can explicitly be specified in the constraint section of the TDL description. The logical formulas are built from the boolean operators conjunction, disjunction, and negation. Atomic expressions are available for specifying parallel execution of operations, operation sequencing and resource restrictions for operands [16].

```
(op1 in {AluOps} & op2 in {MulOps}):
 (op1 && op2) -> ((op1.src1 in {GroupA})
  & (op1.src2 in {GroupB})
  & (op2.src1 in {GroupC})
  & (op2.src2 in {GroupD}));
```

The logical constraints are transformed into integer linear constraints and added to the generated integer linear program. Each logical constraint consists of a premise and a rule. The premise denotes a precondition for the generation of the corresponding ILP constraint that can be statically evaluated. In the example, the ILP constraint will be generated for each pair of operations where the first one belongs to the operation group `AluOps` and the second one to `MulOps`. Those groups have been previously declared in the TDL description. The rule part constitutes a logical implication: if both operations are scheduled on the same control step, the operands must reside in the correct register groups. So the generated ILP constraint in the order-based formulation for two operations i and j is equivalent to the following implication:

$$t_i = t_j \rightarrow \Phi^A_{d_1} = 1 \wedge \Phi^B_{d_2} = 1 \wedge \Phi^C_{d_3} = 1 \wedge \Phi^D_{d_4} = 1$$

The expression $\Phi^g_k = \sum_{(i,k) \in E^{R,g}_F} x^g_{ik}$ represents the register flow of register group g entering an operation k. By forcing the appropriate sum to be equal to one for each operation d_k defining an operand of i and j, it is assured that the operands are located in the correct register groups g. The logical implication is translated into an ILP constraint with the help of binary variables [20,35].

4.5 ILP Models and Hardware Architectures

In our experimental evaluation we could establish a relationship between the architectural design of the target processor and the appropriate ILP formulation style. The flow modelling allows an efficient representation of irregular architectures where the resource competition is high. It also allows an efficient integration of register assignment and instruction scheduling. However if there is a large number of alternative functional units for each operation, the increasing number of alternative resource flows leads to a decrease in solution efficiency. In that case, a time-based formulation is more efficient where the starting time for the operation execution is explicitly bound to control steps. Most ILP formulations for code generation known from literature are time-based formulations. Our results indicate that for irregular architectures, flow-based formulations are more appropriate.

4.6 ILP-Based Approximations

Formulations based on integer linear programming (ILP) allow flexible retargeting of phase-coupled instruction scheduling, resource allocation and register assignment. However the calculation time required for determining an exact solution can grow high. The complexity of integer linear programming is mainly

due to the variables that are explicitly specified as binary. So, the basic idea of the approximations we developed is to iteratively calculate partial relaxations of the original problem where only a subset of the binary variables must take integral values. After each iteration step, the values of the variables considered as binary in this step are fixed to their current value and a new set of variables is specified as binary. The choice of those variables is based on a set of heuristics as presented in [18]. We have shown that by using ILP-based approximations, the computation time can be significantly reduced. The resulting code quality is better than that of conventional graph-based algorithms. A more detailed discussion of those approximations can be found in [18,20].

5 Experimental Results

5.1 Modelled Architectures

Up to now, TDL descriptions have been written for the Analog Devices ADSP-2106X [1], for the Philips TriMedia TM1000 [29], the Infineon TriCore μC/DSP [5] and the Infineon C166 processor. The TDL description of the TriCore processor is used within a framework for calculating WCET guarantees for real-time systems [8]. The C166 description has been used to implement dataflow-based postpass optimisations and is part of a a commercial postpass optimisation tool. TDL descriptions for the TI320C6x [33], and for the Intel Embedded Pentium processor are under construction.

5.2 Optimiser Performance

In order to demonstrate the wide range of supported target architectures, we have evaluated the performance of the optimisers generated for the Analog Devices ADSP-2106X and for the Philips TriMedia TM1000. The ADSP-2106X is a digital signal processor with an irregular architecture and restricted instruction-level parallelism. The TriMedia TM1000 is a VLIW architecture where restrictions on issue slot assignment and result bus synchronisation have to be considered. The integer linear programs have been solved by the CPLEX-library [15]; the calculation times have been measured under SunOS 5.7 on a SPARC Ultra-Enterprise 10000.

The ADSP-2106x SHARC. The optimiser generated for the ADSP-2106X processor performs integrated instruction scheduling and register assignment. The optimiser is generated automatically from the TDL description and allows an optimal solution of those problems with respect to the given code selection; it uses the SILP-based formulation.

In the following we will show experimental results for a set of input programs that are typical applications of digital signal processing: a finite impulse response filter (*fir.s*), a discrete Fourier transformation (*dft.s*), a cascade filter (*casc.s*), and a wavelet transformation (*wave.s*); *wp3.s* is a routine from the whetstone

benchmark. For each program, we compare the result of our optimisations to the optimal code sequence (verified by hand) and to the result of list scheduling algorithms with highest level first heuristics [17]. In order to provide for a fair comparison, an optimal register assignment has been used as an input to the graph-based algorithms (but not to the ILP-based optimisations).

The input programs and their characteristics are shown in Table 1. For each program, the number of operations in the input program, the number of basic blocks and the number of loops is given.

Table 1. Characteristics of the Input Programs (ADSP-2106X).

input	ops	BB	Loops
fir.s	18	3	1
dft.s	26	4	2 (nested)
casc.s	23	4	1
whetp3.s	26	1	0
wave.s	39	10	3 (nested)

In Table 2, the experimental results are summarised. The integer linear programs are solved either exactly or by the use of ILP-based approximations. For reasons of space, only the results of the fastest ILP-based approximation are shown. Details about the ILP-based approximations can be found in [20,18]. All input programs can be represented by a single superblock. For each input program, the column *opt* gives the optimal number of instructions. The solution method is shown in column *mode*; *I1* denotes the exact ILP-based solution, *I2* the ILP-based approximation and *LS* the result of list scheduling with optimal register assignment given. The result of each optimisation method is shown in column *res* as the number of compacted instructions in the output program. The last column indicates the required CPU-time.

For the small input programs, the exact ILP-based solution takes less time than the approximation. This is due to the overhead caused by the approximation setup. However, for larger input programs, the computation time can be drastically reduced compared to the exact solution. In nearly all cases the ILP-based approximations provide an optimal solution. Even with an optimal register assignment given in the input programs, the code produced by the graph-based methods contained on average 19.82% more instructions than the optimal code sequences.

The Philips TriMedia TM1000. The TriMedia TM1000 has 128 general-purpose registers so that the detailed register assignment is less important. However, the instruction scheduling interacts with the slot allocation problem: all operations have to be assigned to issue slots and the assignment of operations to issue slots is restricted. Moreover, the operations have to be synchronised

Table 2. Optimisation Results (ADSP-2106X).

input	opt	mode	res	CPU-time
fir.s	8	I1	8	1.07 s
		I2	8	2.74 s
		LS	9	< 1 s
dft.s	14	I1	14	7.73 s
		I2	14	9.41 s
		LS	14	< 1 s
casc.s	8	I1	8	9.6 s
		I2	8	29.22 s
		LS	12	< 1 s
wp3.s	19	I1	–	> 24 h
		I2	19	10 min 29 s
		LS	24	1.9 s
wave.s	29	I1	29	1 min 55 s
		I2	29	1 min 43 s
		LS	32	< 1 s

with respect to the write-back bus; no more than five operations may write their result simultaneously on the bus. Again, the corresponding optimiser is generated automatically from the TDL description and allows an optimal solution of this problem with respect to the given code selection. Since the integration of register assignment is not needed, both the SILP- and the OASIC-based ILP-formulations can be applied. Our experimental results show that for this processor, the time-based model performs better than the SILP formulation. For reasons of space, only the results of the optimiser using the time-based ILP-formulation are shown. The input programs of the TriMedia TM1000 have been generated with the highly optimising Philips tmcc compiler. The information about the input schedule is not used during ILP solving, so that the scheduling of the input program is undone, and a comparison of the ILP-based methods and the genuine algorithm is possible. The input program *nru.s* is a filtering benchmark, *wp3* is a routine from the whetstone benchmark, *m1x3* is a matrix multiplication and *conv.s* the inner loop of a convolution algorithm. The results are summarised in Table 3. The column *ops* shows the number of operations (nops are not counted), *opt* gives the optimal number of compacted instructions. The number of instructions in the result of each method is shown in column *res* and the required CPU-time is presented in the last column.

Again we can see that for the smaller programs, the exact ILP-based solution can be obtained faster than the approximative result which is caused by the approximation setup time. With increasing program size the calculation time becomes increasingly smaller compared to the exact solution. All ILP-based results could be calculated within some minutes; again the solution quality of the ILP-based approximation is very high.

Table 3. Optimisation Results (TriMedia TM1000).

input	ops	opt	mode	res	CPU-time
ncu.s	33	14	I1	14	7 min 1 s
			I2	15	4 min 49.6 s
			tms	14	–
wp3.s	12	34	I1	34	3.82 s
			I2	34	44.9 s
			tms	34	–
conv.s	42	17	I1	17	5 min 57 s
			I2	17	1 min 25.2 s
			tms	17	–
m1x3.s	31	25	I1	25	4 min 22 s
			I2	26	9 min 17.2 s
			tms	29	–

6 Conclusion and Outlook

The PROPAN system has been presented as a framework that allows for the generation of machine-dependent postpass optimisations and analyses. PROPAN has been especially designed to perform high-quality optimisations for irregular architectures. For each target architecture, a phase-coupled code optimiser is generated from the TDL-specification that can perform integrated global instruction scheduling, register reassignment and resource allocation by integer linear programming (ILP). In the generated integer linear programs, the restrictions and features of the target architecture are precisely taken into account. We have demonstrated the applicability of our approach by modelling two widely used processors with considerably different hardware characteristics. Furthermore we have shown that by using ILP-based approximations, the calculation time can be drastically reduced while obtaining a solution quality that is superior to conventional graph-based approaches. In contrast to most other phase-coupled approaches allowing an exact solution, the optimisation scope is not restricted to basic blocks. The measured calculation times are acceptable for practical use.

There is ongoing work to support complex, superscalar pipelines so that the range of target architectures can be further extended.

References

1. Analog Devices. *ADSP-2106x SHARC User's Manual*, 1995.
2. Siamak Arya. An Optimal Instruction Scheduling Model for a Class of Vector Processors. *IEEE Transactions on Computers*, 1985.
3. S. Bashford and R. Leupers. Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths. *DAES*, pages 1–50, 1999.

4. F. Bodin, Z. Chamski, E. Rohou, and A. Seznec. *Functional Specification of SALTO: A Retargetable System for Assembly Language Transformation and Optimization. rev. 1.00 beta.* INRIA, 1997.
5. E. Farquhar and E. Hadad. *TriCore Architecture Manual.* Siemens AG, 1997.
6. A. Fauth, J. Van Praet, and M. Freericks. Describing Instruction Set Processors Using nML. In *Proceedings of the EDAC*, pages 503 – 507. IEEE, 1995.
7. C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems.* PhD thesis, Saarland University, 1997.
8. C. Ferdinand, D. Kästner, M. Langenbach, F. Martin, M. Schmidt, J. Schneider, J. Theiling, S. Thesing, and R. Wilhelm. Run-Time Guarantees for Real-Time Systems - The USES Approach. *Proceedings of the ATPS*, 1999.
9. J.A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, pages 478 – 490, 1981.
10. C.H. Gebotys and M.I. Elmasry. Global Optimization Approach for Architectural Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1266 – 1278, 1993.
11. R. Govindarajan, Erik R. Altman, and Guang R. Gao. A Framework for Resource Constrained Rate Optimal Software Pipelining. *IEEE Transactions on Parallel and Distributed Systems*, (11), 1996.
12. G. Hadjiyiannis. ISDL: Instruction Set Description Language Version 1.0. Technical report, MIT RLE, 1998.
13. A. Halambi, P. Grun, V. Ganesh, Khare A., N. Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. *DATE*, 1999.
14. Silvina Hanono and Srinivas Devadas. Instruction Scheduling, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *Proceedings of the DAC*. ACM, 1998.
15. ILOG S.A. *ILOG CPLEX 6.5 User's Manual*, 1999.
16. D. Kästner. TDL: A Hardware and Assembly Description Language. Technical Report TDL1.4, TFB 14, Saarland University, 2000.
17. D. Kästner and M. Langenbach. Integer Linear Programming vs. Graph-Based Methods in Code Generation. Technical report, Saarland University, 1998.
18. D. Kästner and M. Langenbach. Code Optimization by Integer Linear Programming. In *Proceedings of the CC*, pages 122 – 136, 1999.
19. D. Kästner and S. Thesing. Cache Sensitive Pre-Runtime Scheduling. In *Proceedings of the LCTES Workshop*, 1998.
20. Daniel Kästner. *Retargetable Code Optimization by Integer Linear Programming.* PhD thesis, Saarland University, 2000. To appear.
21. Kästner, D. and Wilhelm, R. Operations research methods in compiler backends. *Mathematical Communications*, 1999.
22. M. Langenbach. CRL – A Uniform Representation for Control Flow. Technical Report CRL1, TFB 14, Saarland University, November 1998.
23. Rainer Leupers. *Retargetable Code Generation for Digital Signal Processors.* Kluwer Academic Publishers, 1997.
24. R. Lipsett, C. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, 12. edition, 1993.
25. Peter Marwedel and Gert Goossens. *Code Generation for Embedded Processors.* Kluwer, 1995.
26. S. Novack and A. Nicolau. Mutation scheduling: A Unified Approach to Compiling for fine-grain Parallelism. In *Languages and Compilers for Parallel Computing*, pages 16–30. Springer LNCS, 1994.

27. L. Nowak. Graph Based Retargetable Microcode Compilation in the MIMOLA Design System. *20th Annual Workshop on Microprogramming*, pages 126 – 132, 1987.
28. S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA: Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. *Proceedings of the DAC*, 1999.
29. Philips Electronics North America Corporation. *TriMedia TM1000 Preliminary Data Book*, 1997.
30. John Ruttenberg, G.R. Gao, A. Stoutchinin, and W. Lichtenstein. Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler. *Proceedings of the PLDI*, pages 1 – 11, 1996.
31. M.A.R. Saghir, P. Chow, and C.G. Lee. Exploiting Dual Data-Memory Banks in Digital Signal Processors. *Proceedings of the ASPLOS*, 1996.
32. Ashok Sudarsanam. *Code Optimization Libraries For Retargetable Compilation For Embedded Digital Signal Processors*. PhD thesis, University of Princeton, 1998.
33. Texas Instruments. *TMS320C62xx Programmer's Guide*, 1997.
34. Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.
35. H.P. Williams. *Model Building in Mathematical Programming*. John Wiley and Sons, 1993.
36. L. Zhang. *SILP. Scheduling and Allocating with Integer Linear Programming*. PhD thesis, Saarland University, 1996.
37. V. Zivojnovic, J. M. Velarde, C. Schläger, and H. Meyr. DSPSTONE: A DSP-Oriented Benchmarking Methodology. In *Proceedings of the International Conference on Integrated Systems for Signal Processing*, 1994.

A Framework for Enhancing Code Quality in Limited Register Set Embedded Processors

Deepankar Bairagi, Santosh Pande, and Dharma P. Agrawal

Department of Electrical and Computer Engineering and Computer Science
University of Cincinnati
Cincinnati, OH 45221
email: dbairagi, santosh, dpa@ececs.uc.edu

Abstract. Spill code generated during register allocation greatly influences the overall quality of compiled code, both in terms of speed as well as size. In embedded systems, where size of memory is often a major constraint, the size of compiled code is very important. In this paper we present a framework for better generation and placement of spill code for RISC-style embedded processors. Our framework attempts to achieve efficient execution and reduce spill-induced code growth. Traditional graph-coloring allocators often make spilling decisions which are not guided by program structure or path-sensitive control flow information. Quite often, allocation decisions get heavily influenced by the choice of candidates for register residency. Especially for systems with a limited number of registers, if one is not careful to contain register pressure, it could lead to generation of a lot of spill code. We propose a framework which selectively demotes variables in a contained manner and influences the formation of live ranges. The decisions for selective demotion are made through a flow-analytic approach such that fewer spill instructions are generated. Our approach tries to keep variables as candidates for register allocation only along the paths where it is profitable to do so. We attempt to identify good local candidates for demotion, however, decisions are taken only after their global demotion costs are captured. We have implemented our framework inside the SGI MIPSPRO compiler. Our results show improvement over a Briggs-style allocator in reducing code size upto 3.5% and upto 8.2% in reducing static loads in some cases for a register set of size 8. The results are very encouraging for other parameters as well for various sizes of register sets.

1 Introduction

Embedded processors are used in a wide variety of applications ranging from cell phones to automobiles. Efficient programming for embedded processors is a challenging problem due to the stringent requirements like limited memory, availability of fewer registers and real time response. Of late, with the arrival of larger applications, assembly language programming for such systems have given way to programming in high level languages. However, the requirements

J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 81–95, 2001.

for embedded systems render many of the traditional optimizations useless. Owing to tight memory constraints, code size is an extremely critical issue while generating code for embedded processors. Code size has direct impact on issues such as cost, power consumption, transmission time, etc. in embedded systems. Traditional compiler optimizations are often geared towards speed rather than size for compiled code. Register allocation is a phase which encompasses these often orthogonal and sometimes conflicting goals addressed during code generation for embedded systems. A good amount of code generated for any processor comprises load and store instructions and many of these loads and stores are spill code generated during register allocation. This observation is particularly true for RISC processors and, lately, such processors have found widespread use in the domain of embedded systems due to advantages such as code density etc. Therefore, decisions made during register allocation can noticeably impact the code size for such processors. The effect of spill code on code size is even more conspicuous for processors with limited (8 or 16) registers, and the allocation decisions need to be really frugal and smart while using the limited set of registers in order to avoid excessive spilling. Our framework focuses on assisting a register allocator to make good decisions such that generation of spill code is minimized.

The case for "speed" is a goal that embedded processors share with general-purpose processors, albeit with a different degree of importance. In general, lesser spill code can mean fewer instructions to execute, therefore, faster execution. Memory access instructions like loads and stores are usually of variable latency due to cache misses etc. Presence of such instructions can significantly slow down the execution of any instruction sequence. And such instructions are more likely to be encountered in code generated for limited register set embedded processors due to higher register pressure. Optimizations like PRE, which attempt to reduce run-time overhead of computing redundant expressions, can also contribute to increase in register pressure by introducing new temporaries. Embedded systems typically have smaller instruction caches. In such systems increase in code size can have direct negative effect on the "speed" factor during execution. Load and store instructions can also negatively influence the quality of an instruction schedule by inhibiting the movement of instructions. Therefore, it is imperative that the register allocator does a good job of assigning registers to program variables and temporaries.

Register allocation strategies, such as graph coloring [2,3,5,6,11], neatly capture the concept of simultaneous liveness or lack thereof. Such strategies provide a reasonably good allocation at a moderate cost when the demand for registers is not very high. However, they fall short of providing an adequately efficient global allocation of registers when demand for registers is high and spills are forced. With a limited set of available registers, as is the case in many embedded processors, it is worthwhile spending more time doing a better allocation. For, programs running in embedded systems are usually compiled only once and sometimes burned into ROM. While the number of loads and stores is a fairly good indication of the size of the generated code, the placement of the loads and

stores is often as important a factor in dictating performance. Other compiler optimizations like CSE, which could possibly reduce code size, also depend on the register allocator to find registers for temporaries that are generated during the optimization phase. The efficacy of such expensive optimizations are lost if the temporaries are spilled to memory. In this work, we focus on issues related to choice of candidates for register allocation.

1.1 Register Pressure and Register Candidates

Besides coloring or spilling decisions, register allocation is closely linked with decisions made during *register promotion*. Register promotion identifies the program variables which are safe to be kept in registers and the program segments over which a particular variable can be safely kept in registers. Even though register promotion assists register allocation by placing program variables in pseudo registers, the job of the register allocator is far from trivial. Forced spills not only undo the job performed through register promotion earlier, but also can lead to bad spilling decisions and non-optimally placed spill code. Thus, register promotion sometimes works against good allocation by promoting far too many variables and by increasing register pressure beyond a reasonable limit. Our approach, therefore, is centered around the idea of containing register pressure such that an allocator can make good decisions. Live ranges of variables promoted to pseudo registers constitute the candidates for register allocation. Profitability of register promotion, particularly in the context of register pressure, is usually not addressed because promotion is carried out as a machine independent optimization. Therefore, our approach tries to identify areas of high register pressure in a program and judiciously selects variables to be demoted around such areas so that a balanced register pressure is maintained.

1.2 Limitations of Existing Approaches

In majority of the graph coloring allocators, the spilling strategy is "all-or-nothing". Once a node is spilled, it is spilled everywhere. However, spilling should be carried out only across the regions of high register pressure and only those variable with no or fewest uses along those regions should be spilled. Most of the approaches are not sufficiently sensitive to program structure. Hierarchical approaches [4,14] try to overcome this drawback by performing register allocation over portions of code separately and then by merging these allocation decisions over the entire program or routine. An obvious benefit of doing so is that a variable can possibly be spilled only locally. The hierarchical approaches do address the issue of spilling less frequently used variables around regions by giving priority to local variables. However, spilling decisions are performed early in the allocation phase and variables which may appear to be good candidates for spilling in the bottommost hierarchy may introduce spill code which is inefficient from the viewpoint of the complete program.

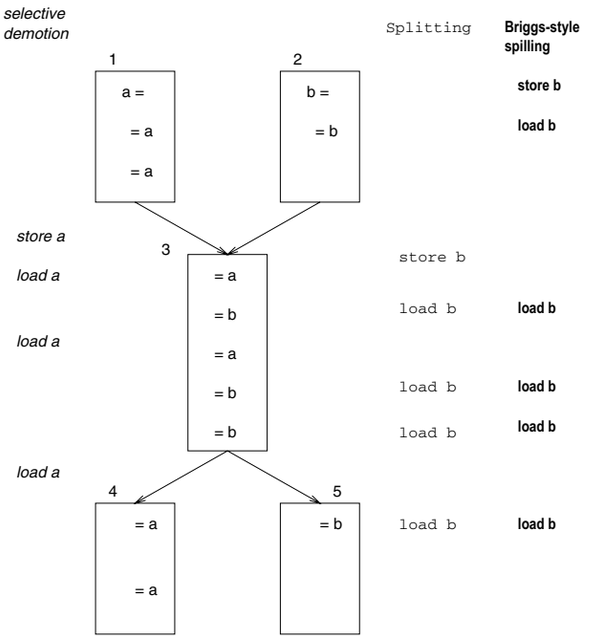


Fig. 1. Traditional spilling or splitting

1.3 Motivating Example

In Figure 1 we illustrate the limitations of standard algorithms with an example. We assume that basic block 3 has high register pressure and one of the two variables `a` or `b` needs to be spilled.

Splitting: `a` has 6 uses as opposed to `b` which has 5 uses. Therefore `a` has higher priority than `b` and is allocated a register. The splitting algorithm then recognizes that `b` can still be allocated a register in basic block 2. Thus, the live range of `b` is split, allocated a register in basic block 2, and spilled in basic block 3.

Briggs-style spilling: `b` has lower spill cost than that of `a`, and therefore, is spilled entirely over basic blocks 2-3-5.

However, more efficient code is generated if `a`, which has fewer uses in the high register pressure region (basic block 3), is spilled. Selective register demotion attempts to achieve this by demoting `a` in basic block 3 before the register allocator attempts coloring.

In figure 2, we show a small segment of code and an ideal allocation of registers. Assuming that there are three registers available, either `x` or `y` needs to be spilled, and should be spilled only along the high register pressure region. Variables `a, b, c, d` have sufficient uses so that they are not considered for spilling. In this example we choose to spill `x` (`y` is as good a candidate) and the spill code is inserted around basic block 2. In figure 2 (a), we show how a

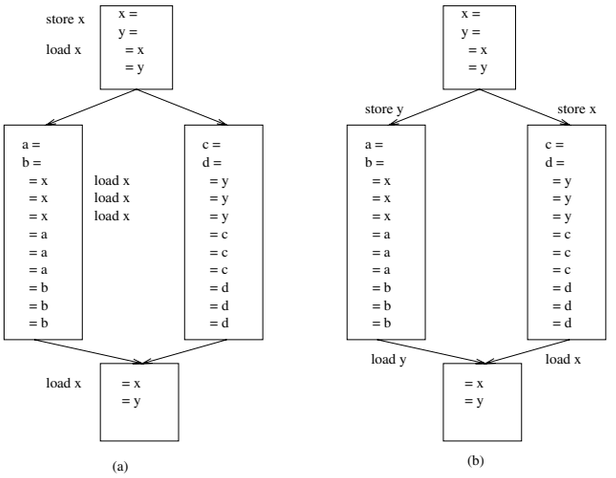


Fig. 2. Control-flow-sensitive allocation

Briggs-style allocator would spill the variable `x`. `x` is spilled over its entire live range. Our objective is to reach an allocation as shown in figure 2 (b). `x` and `y` are demoted along the boundaries of the regions where they are profitable to be considered for register allocation. As obvious, we reduce the number of spill code from 6 to 4. However, the run-time gains of such an allocation is also significant since only 2 spill instructions are executed in 2 (b).

In this paper, we present a framework for global register allocation based on *selective demotion*. We propose a flow-analytic approach wherein global allocation of registers is guided by judicious placement of loads and stores along the different paths of control-flow in the program to improve overall code quality. We do not seek to build a new allocator; however, we attempt to positively influence an underlying graph-coloring allocator by augmenting it with a phase of selective demotion. In situations where register allocation forces spill of variables, we achieve more efficient allocation by *selective demotion* of program variables along regions of high register pressure. Variables are considered for allocation only along those paths of control-flow where it is profitable to keep the variable in registers. The global information about profitability or spill cost is fed into the register allocator by influencing the formation of live ranges, and thus guiding the allocator to generate globally efficient spill code. We also minimize the effect of spill code by placing it outside regions of heavy execution frequency whenever possible. The novelty of our approach is highlighted by the following points that we address:

- The idea of spilling or splitting certain variables or their live ranges before a graph coloring allocator.
- The way the spilling or splitting is performed -
  - Identification of the variables to be spilled or split.

- Identification of the program segments over which this is carried out.

We limit our analysis to register allocation within procedure boundaries. The rest of the paper is organized as follows. In the next section, we discuss our approach in detail. In the third section, we discuss implementation issues and results. In the fourth section we discuss some of the issues related to code size in embedded systems and register allocation in general. In the last section, we draw conclusions and discuss future work.

2 Our Approach

In this section we discuss our framework of selective demotion. Selective demotion attempts to contain register pressure such that allocators for processors with a limited set of registers can make smart decisions about spill code generation. The crux of our approach is on selectively demoting variables around regions of high register pressure:

- heavy use favors promotion
- sparse use favors demotion

The promotion phase has already promoted variables optimistically and placed them in pseudo-registers. The demotion phase is carried out as a pre-pass of actual global register allocation (GRA). The effect of demotion of a particular variables is manifested as splitting of its live range before allocation and spilling part of its range over certain basic blocks. Some of the known register allocators attempt to optimize the placement of spill code or try to move them out of regions of high execution frequency as a post-pass of GRA. However, spilling decisions are already made and it may not be possible to move spill code thus introduced. The major benefit obtained by our approach is that unlike trying to repair the damage of non-optimal spill code, we try to avoid making such decisions by judiciously demoting variables. In the underlying framework, similar to hierarchical approaches, we start at the basic block granularity to identify the candidates for demotion and try to merge or propagate the areas of demotion into larger regions. However, unlike the Tera [4] or RAP [14] allocator, we delay the actual demotion decision till global profitability has been captured. Following this, the register allocator takes over and works on the modified live ranges.

2.1 High Register Pressure Region

The first step of our framework consists of identification of the areas of high register pressure. Compared to traditional general purpose architecture, register pressure is expected to be higher in embedded processors with a small register set. For every basic block, we construct the sets reg_candidate for variables or temporaries which are already in pseudo registers and expected to be in a register in that basic block. The reg_candidate information is essentially *live range* information, calculated at a per basic block granularity. It is the set of live ranges

passing through that basic block. The reg_candidate information is gathered by using traditional data-flow analysis to compute *liveness* and *reaching-definition* attributes [1]. The analysis is similar to the one performed in the Chow allocator [6].

A variable `v` is *live* at a basic block `B` if it is referred at B or at some succeeding block without preceded by a redefinition. Global *liveness* information is computed by backward iteration over the control flow graph.

A definition `d` of a variable `v` is *reaching* a basic block `B` if a definition or use of the variable reaches B. Global *reaching-definition* information is computed by forward iteration over the control flow graph.

The intersection of the two attributes, that is, if a variable is both live and reaching a basic block, gives whether the variable is a candidate at the basic block. These variables were already placed on pseudo registers earlier and will be considered for register allocation in that block by the succeeding allocation phase. Thus, for every basic block B, we have a set reg_candidate[B] saying which variables are candidates in that block. And for every variable `v` v we also construct a set reg_candidate_in saying in which blocks it is a candidate.

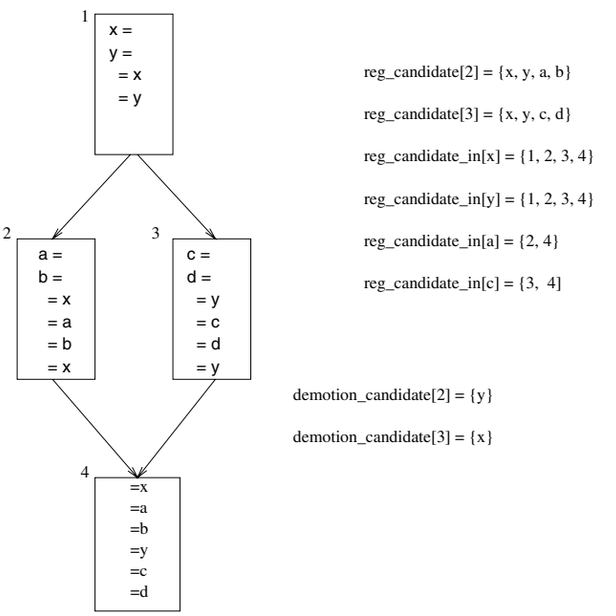


Fig. 3. Example of reg_candidate attribute

Figure 3 illustrates the reg_candidate and reg_candidate_in information in a simple example. The cardinality of the reg_candidate set gives the *register_pressure* of the basic block. Assuming there are k physical registers

available for allocation, a basic block with $register\_pressure > k$, is marked as a high register pressure region.

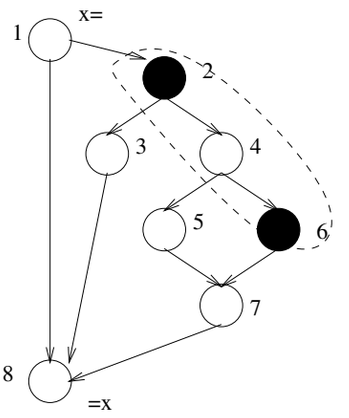
2.2 Demotion Candidates

For every region of high register pressure, we identify the variables which are promoted through the region and have the least number of uses or definitions. These variables constitute the initial set of demotion candidates for that particular basic block. We try to identify at least *register_pressure* - `k` initial candidates. In figure 3, we show the initial demotion candidates for basic blocks 2 and 3 where register pressure is 4 (assuming there are 3 available registers). We also compute the tentative estimate of demotion cost if a particular variable were to be demoted across the region. Many of the register allocators use a spill cost estimate given by $(\#loads + \#stores).10^{depth}$, assuming so many additional loads and stores will be inserted as a result of spilling and *depth* indicates the depth of loop nesting. Our estimation of demotion cost is similar except that it is weighted by the static frequency estimate of that basic block. The weighing factor is a number between 0 and 1 indicating the likelihood of the basic block being executed. The particular basic block forms the initial *tentative* demotion region for a demotion candidate. In the following discussion, a demotion region is a *tentative* demotion region unless otherwise specified. We assume that besides a load preceding every use of a variable, demotion of a variable in a particular block would result in a store in the entry of the basic block and a load in the exit of the basic block.

2.3 Merger of Demotion Regions

Following the initialization of the demotion candidates, we attempt to enlarge the demotion regions for the candidates, if it leads to better overall demotion cost. In particular, we attempt to merge two contiguous or disjoint demotion regions for a particular variable if it happens to be a demotion candidate for both the regions. A merger would lead to the elimination of a load and a store at the edge connecting the merging regions. In order to maintain safety, every exit point of a demotion region has a load for the particular variable under consideration. These mergers of demotion regions are final, however, the decision whether the particular variable will be demoted at all is taken later. We illustrate with a simple example in figure 4 how merger of demotion regions may lead to better allocation.

The shaded nodes in the control flow graph are high register pressure regions. `x` is a demotion candidate in both the nodes 2 and 4. The dotted region shows the merged demotion region 2-4-6. The numbers in the second and third column indicate the number of loads and stores that would be executed if the control follows any one of the paths as represented by the first column. As obvious, in this particular example, merger of the demotion regions is always as good as demoting them only in the blocks 2 and 4. If there were more uses of x in nodes 1 and 8, demotion in 2 and 6 or 2-4-6 far outweigh any benefit of spilling x



	2 & 6	2-4-6
1-8	0	0
1-2-3-8	2	2
1-2-4-5-7-8	2	2
1-2-4-6-7-8	4	2

Fig. 4. Merger of demotion regions

all over place. However, merger of demotion regions is not always profitable. If there were multiple uses of x in node 4, demotion along 2 and 6 would be more profitable than demoting along 2-4-6. If there was just one use of x in node 4, it is still worthwhile demoting along 2-4-6. For, it may be possible to enlarge the demotion region further and move the spill code at the entry and exit out of a loop if such a situation exists.

For a given demotion candidate, merging of demotion regions is attempted along any path joining two demotion regions. If the static estimate of demotion cost for the merged region is cheaper than the sum of the two individual demotion regions, the merger is carried out. Otherwise, merger of the two regions for that variable is aborted. As obvious from the discussion, if there are several uses or definitions of a variable along a path, the estimate for demotion will go up and it will eventually prevent the variable from being demoted. This is desirable since demotion is deemed beneficial only for global variables with sparse or no uses along a particular path or region.

2.4 Demotion Decisions

Having determined the various demotion regions, we sort the spill cost estimates associated with each demotion region. We also keep a count of the number of high register pressure regions a demotion region spans. A good candidate for demotion

Table 1. Results: Mediabench with 16 registers

program	demotion	code size	static loads	static stores	dynamic instr. count
epic	ON	141032	3452	1816	88667017
(epic+unepic)	OFF	141480	3481	1867	90776178
ghostscript	ON	2656160	100317	46625	947436917
	OFF	2724616	107873	46936	966891145
mpeg2encode	ON	179656	7139	2095	1393841237
	OFF	181280	7141	2110	1412219693
mpeg2decode	ON	167080	4979	1479	105943012
	OFF	168028	5111	1524	109049201
g721	ON	72076	508	270	698904843
	OFF	72912	530	279	706381869

is one which has the least demotion cost estimate and spans most number of high register pressure regions. Guided by this heuristic, we demote the best possible candidate. Then we check if register pressure has gone down sufficiently or candidate demotion regions are still available. This process is continued till we run out of demotion region candidates or register pressure has gone down sufficiently across the program. This is the phase when loads and stores are inserted.

The steps discussed above may appear to be costly or time-consuming. However, such a situation is likely only if in a fairly complex control-flow graph, the high register pressure region with a candidate x are scattered far apart with very sparse or no use at all. Nearby high register pressure regions are the ones which are most likely to be merged successfully. And merger with the further ones are likely to be given up if sufficient uses are encountered on the way.

The procedure we describe above will work with any kind of demotion candidates so far as merging them along a sparse-use path is concerned. The idea guiding our demotion algorithm is to gather the global view of the spill cost associated with locally good candidates.

3 Implementation and Results

Our framework is implemented within the backend of the MIPSPRO compiler from Silicon Graphics (SGI). The code generator of the MIPSPRO compiler comprises a framework for global scheduling called Integrated Global Local Scheduling (IGLS) [13]. Global Register Allocation (GRA) is preceded by a round of global and local scheduling. Our framework is implemented following this round scheduling as a frontend to GRA. GRA is followed by another round of global and local scheduling.

Table 2. Results: Mediabench with 12 registers

program	demotion	code size	static loads	static stores	dynamic instr. count
epic	ON	143596	3591	1848	101532118
(epic+unepic)	OFF	144168	3621	1909	10372928
ghostscript	ON	2681444	108316	50117	958821847
	OFF	2760826	116493	50491	978726411
mpeg2encode	ON	185146	7386	2243	1465029931
	OFF	186798	7392	2267	1494331552
mpeg2decode	ON	1789622	5208	1593	108831095
	OFF	1804054	5349	1644	113769238
g721	ON	73942	550	276	766380941
	OFF	74934	574	285	779855204

Table 3. Results: Mediabench with 8 registers

program	demotion	code size	static loads	static stores	dynamic instr. count
epic	ON	149788	3889	2009	128819591
(epic+unepic)	OFF	150536	3922	2087	131285837
ghostscript	ON	2768092	117175	58718	971678741
	OFF	2870208	126834	59245	995402151
mpeg2encode	ON	196472	7944	2695	1603150746
	OFF	198204	7964	2829	1649281656
mpeg2decode	ON	195076	5565	1860	114284351
	OFF	198204	5727	1929	122405289
g721	ON	77428	610	302	894566070
	OFF	78584	644	312	919440707

Tables 1, 2 and 3 show the improvement in various parameters in test runs of a few programs from the Mediabench suite of benchmarks (from UCLA). Application programs from Mediabench are quite commonly used as embedded applications. The comparison is with a Briggs-style allocator with selective demotion turned on and off. We have carried out the experiments on an MIPS RISC R12000 processor. We have implemented the Briggs-style allocator. Various allocators also work on different levels of intermediate representation and depend upon the optimizations performed before and after allocation. We have attempted our best to provide a fair comparison with our Briggs-style allocator.

As the numbers indicate, the code quality is never worse than the underlying allocator. A general observation is that there is room for improvement when the register pressure is fairly high. Ghostscript is considerably larger than the other benchmarks, and as the number of static loads and stores indicate, the improvement ranges from 7.5% for a set of 16 registers to 8.2% for a set of 8 registers. Loads and stores constitute a good amount of the actual instructions executed and any improvement in their generation and placement also improves the dy-

namic instruction count. Dynamic instruction count improvements indicate that selective demotion leads to better runtime performance as well. Although a much smaller program, mpeg2decode shows considerable improvement in dynamic instruction count ranging from 3.8% for a set of 16 registers to 7.0% for 8 registers. However, the improvement in code size is not expected to be of the same proportion as that of static loads and stores unless the loads and stores dominate the total instructions generated and executed. The same applies for improvements in dynamic instruction count as well. The improvement in static loads and stores as well as code size is more visible when generating code for 8 registers. This is expected since more spills are likely with fewer registers. Ghostscript shows improvements ranging from 2.6% through 3.9% in code size. The numbers reported here are the absolute counts of loads and stores. It is not a count of loads and stores only due to spills.

4 Related Work

In this section we discuss some relevant work in the areas of code size reduction for embedded processors and graph coloring register allocation.

4.1 Code Size Reduction

A good body of research work has been directed at reducing code size for embedded processors. Many of these approaches attempt to compress already compiled code at binary or assembly level. Some strategies apply data compression to executables on disk, and then decompress while loading the binaries onto memory [9]. Another strategy loads compressed executable onto memory and then applies decompression during program execution [16]. Fraser *et al.* uses a suffix tree to find repeated code sequences [10]. Cooper *et al.* [7] extend Fraser's work and use pattern-matching techniques to identify and coalesce together repeated instruction sequences.

Liao *et al.* [12] delays storage allocation to the code generation phase, thus increasing use of specific address modes like auto-increment/decrement. This results in reduction of address arithmetic leading to reduction in the size of generated code. Rao *et al.* [15] extends this work by presenting heuristic techniques to optimize the access of variables by applying algebraic transformations on expression trees. Yet another approach attempts to decrease process memory requirements by overlapping uninitialized global variables with other static data. Traditional compiler optimizations have significant impact on code size although these transformations are targeted towards speed more than size. The application sequence of such optimizations e.g., constant propagation, dead-code elimination, strength-reduction, lazy code motion, etc., have different impact on code size depending upon the order of these applications. Another strategy by Cooper *et al.* [8] uses genetic algorithms to determine a good optimization sequence.

4.2 Register Allocation

Most of the work on register allocation as graph coloring are variants of or extensions to Chaitin's or Chow's work [5,6]. Chaitin's graph coloring approach [5] is based on the concept of an *interference graph.* The nodes of an interference graph represent the *live ranges* of the variables in a program. Two nodes of an interference graph are connected by an edge if the corresponding live ranges are simultaneously live. Given `k` physical registers, register allocation is reduced to `k`-coloring the interference graph. Nodes with *degree* <`k` are guaranteed to find a color. Such a node is termed an *unconstrained* node. All the unconstrained nodes and their edges are removed from the graph and pushed down a stack for making coloring decisions later. If the resulting graph is empty, then the original interference graph can be trivially colored. Once the coloring blocks, the allocator attempts to reduce the the degree of the interference graph by *spilling* a live range. A spilled live range is assigned to memory and the corresponding node is removed from the interference graph. If the spill frees up some other constrained nodes, the algorithm proceeds as above by putting those nodes in the stack. Otherwise, it continues to spill till the interference graph is empty. Nodes in the stack are then assigned colors so that no two neighbors have the same color.

Briggs [3] suggests an *optimistic spilling heuristic* which postpones the actual insertion of spill code for a node marked for spilling. Such a node is removed from the interference graph, however, it is pushed in the same stack as the unconstrained nodes. Later, when nodes are removed from the stack, it is likely that there could still be a color left to color the constrained node. Briggs's heuristic exposes one weakness of Chaitin style approaches - interference graph may give an excessively conservative view of the "simultaneously live" information. Briggs's allocator always performs at least as good as Chaitin's and sometimes performs better.

spill_cost/degree is a criterion which often guides the choice of a spilling candidate. Another commonly used spilling heuristic is that - if the use of a live range is easy to recompute, it should be recomputed rather than reloaded.

A few approaches [2,6] attempt to spill only along the interfering regions. While that is an improvement, all of them suffer from a basic limitation - *the choice of which variables should be kept in registers and which variables should be spilled is going to be different along different paths of control-flow in the program.* Callahan et al. [4] describe an allocator for the Tera compiler which builds a *tile* tree to represent the control-flow structure of the program. Tiles are a hierarchical structure with basic blocks at the bottommost level of hierarchy. Tiles are visited in a bottom up fashion, allocating variables to virtual registers by graph coloring. This information is passed on to parent tiles and eventually virtual registers are allocated to the whole program. Then the tiles are visited in a top-down fashion and virtual registers are bound to physical registers. The top down pass also introduces spill code or shuffle code as and when necessary. Norris et al. [14] describe a program dependence graph (PDG) based register allocator named RAP. The hierarchy of program segments is represented by

region nodes in the PDG. The difference with the Tera compiler, besides the use of region-structure for tile-structure, is the time when actual binding of physical registers to virtual registers takes place and the way spill code is inserted. The Tera compiler builds interference graphs on the way up and on the way down. Assignment of physical registers takes place on the second phase while spilling decisions are already made during the first phase. RAP, however, performs the assignment of physical registers as well as insertion of spill code in the first bottom up phase itself. During the top-down phase, RAP tries to move spill code out of loops.

5 Conclusion

In this paper we have presented a framework which attempts to achieve faster execution and reduce spill code related code growth for limited register set embedded systems. Our framework of selective demotion assists register allocation by keeping down register pressure and thereby guiding the allocator to generate fewer spill code. Selective demotion chooses variables which are less likely to be used along a certain path and inserts loads and stores at strategic points to reduce the demand for registers. The choice is carried out by identifying locally good candidates for demotion and by capturing their global cost of demotion. Thus, we achieve the goal of splitting live ranges of variables such that spilling happens only along the areas of high register pressure and thus leads to more efficient generation and placement of spill code.

The experimental results demonstrate that selective demotion can positively influence a register allocator to achieve noticeable impact on code size as well as speed. The empirical data also supports our observation that reduction in spill code favorably reflects on the overall goal of code size reduction. Improvement in dynamic instruction count also indicates that fewer instructions are executed as a result of better generation and placement of spill code. The additional time spent performing selective demotion is worthwhile for limited register set processors and code size reduction and instruction count improvement is more pronounced as the size of register set decreases.

We are currently working on gathering results about efficacy of our framework on a splitting allocator.

6 Acknowledgement

We would like to thank SGI for allowing us to use the MIPSPRO backend compiler.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Priciples, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

2. P. Bergner, P. Dahl, D. Engebretsen, and M. O'Keefe. Spill Code Minimization via Interference Region Spilling. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 287–295, June 1997.
3. P. Briggs, K. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.
4. D. Callahan and B. Koblenz. Register Allocation via Hierarchical Graph Coloring. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 192–203, June 1991.
5. G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register Allocation via Coloring. *Computer Languages*, 6:47–57, January 1981.
6. F. Chow and J. Hennessy. The Priority-based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.
7. K. Cooper and N. McIntosh. Enhanced Code Compression for Embedded RISC Processors. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 139–149, May 1999.
8. K. Cooper, P. Schielke, and D. Subramanian. Optimizing for Reduced Code Space using Genetic Algorithms. In *Proceedings of the 1999ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embdedded Systems*, pages 1–9, July 1999.
9. J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting. Code Compression. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 358–365, June 1997.
10. C. Fraser, E. Myers, and A. Wendt. Analyzing and Compressing Assembly Code. *SIGPLAN Notices*, 19(6):117–121, June 1984.
11. P. Kolte and M. J. Harrold. Load/store Range Analysis for Global Register Allocation. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 268–277, June 1993.
12. S. Liao, S. Devadas, K. Keutzer, S. Tijang, and A.Wang. Storage Assignment to Decrease Code Size. *ACM Transactions on Programming Languages and Systems*, 18(3):684–691, May 1996.
13. S. Mantripragada, S. Jain, and J. Dehnert. A New Framework for Integrated Global Local Scheduling. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 167–175, October 1998.
14. C. Norris and L. L. Pollock. RAP: A PDG-based Register Allocator. *Software - Practice and Experience*, 28(4):401–424, April 1998.
15. A. Rao and S. Pande. Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded DSPs. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 128–138, May 1999.
16. A. Wolfe and A. Chanin. Executing Compressed Programs on an Embedded RISC Architecture. In *Proceedings of the 25th IEEE/ACM International Symposium on Microarchitecture*, pages 81–91, December 1992.

A Stochastic Framework for Co-synthesis of Real-Time Systems

S. Chakraverty[1] and C.P. Ravikumar[2]

[1] Netaji Subhas Institute of Technology,
Division of Computer Engineering, New Delhi-110045, India,
sc_12@hotmail.com
[2] Indian Institute of Technology,
Department of Electrical Engineering, New Delhi-110016, India,
rkumar@ee.iitd.ernet.in

Abstract. *In this paper, we propose a stochastic model for hardware software cosynthesis of a heterogeneous, multiprocessor computing system dedicated for a specified real time application. In this model, the task execution times and the data transfer times are taken to be random variables. Based on the stochastic framework, we derive a method for generating optimum task schedules and evaluating the performance of the architecture. The pool of resources required for building the architecture and the task allocations are optimized by a genetic algorithm. We demonstrate that this approach produces architectures which are superior in terms of cost and processor utilization. Moreover, it yields good solutions even in situations where no feasible solution could be produced using deterministic timings. The scheduling algorithm has a polynomial time complexity. The components of the architecture are evolved in a hierarchical manner, progressively refining it by applying the genetic algorithm in distinct phases. This provides a powerful CAD tool for cosynthesis which can generate a range of optimum solutions with exchangeable cost and performance benefits.*

1 Introduction

The design of real time systems which fall into the *soft* category can benefit from the fact that such systems do not require a very high degree of discipline regarding meeting the deadlines specified for the system [11]. For example , consider a robot controlled system which picks up and analyzes objects passing on a conveyer belt, and puts them in separate boxes according to their shapes. While the robot controller should be able to classify the objects in synchronism with the frequency with which they arrive, an occasional miss can be dealt with in the next pass. In many applications such as video teleconferencing, buffering and interpolation techniques are used to predict an output (image) whenever there is an occasional miss , but frequent deadline misses gradually lead to unacceptable levels of performance degradation. It therefore makes greater sense to talk of the probability with which the deadlines are missed, and the design of the underlying computing system should be geared towards reducing it .

J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 96–113, 2001.

Current research in the area of hardware software codesign is based on the assumption that the execution times are deterministic quantities [9],[3],[13],[4], [5],[7], [12]. In [13], Prakash and Parker propose Mixed Integer Linear Programming(MILP) for co-synthesis. In [4] and [5], Dave and Jha make use of heuristic task-clustering as a basis for generating sub-architectures. Hou and Shin, in [8], employ the branch and bound method to map and schedule tasks on an existing distributed architecture. In [7] Ravikumar and Gupta employ a genetic algorithm to map tasks onto a parallel processor architecture. V. Nag proposes a genetic algorithm to generate fault tolerant architectures in [12]. It is thus evident that a variety of optimization methods for the cosynthesis problem have been investigated. However, none of these papers take into account the random variations of timing parameters. We adopt a stochastic framework ,whereby the execution times as well as communication times are assumed to be random variables.

Some studies on the stochastic approach for execution times has been reported in the literature. In [1], Li and Antonio have used numerical simulations to estimate the overall execution time distribution of a task graph. Min Tan and Siegel have developed data relocation heuristics in a distributed system based on a stochastic model in [14]. However, the issues of generating optimum task allocations and schedules have been ignored; an architecture is assumed to pre-exist. In contrast, we have defined all aspects of co-synthesis in the light of the proposed stochastic model. We have taken into account the randomness of communication times, which has hitherto been ignored. We present the results of plugging a stochastic scheduling algorithm into the evolutionary search technique of genetic algorithm for the co-synthesis optimization problem.

The rest of the paper is organized as follows. In Section II, the theoretical foundation of the stochastic approach is given. Section III presents the working of the stochastic model in the context of co-synthesis. The adaptation of genetic algorithm for optimization is described in Section IV. Results and their analyses are given in Section V. We conclude in Section VI.

2 Theoretical Foundation

The randomness in task execution times arise due to factors such as variations in data volumes, values of loop control variables, outcome of conditional branches etc. However, real time functions are usually designed to curb infinite loops and asymptotically growing processes. This ensures that there are fixed upper and lower bounds for the execution times. Therefore, we assume the task execution times and the volumes of data transferred between communicating tasks to possess *beta distributions*. Beta distributions can model a wide range of bounded distributions including the uniform distribution. The reader is referred to [6] for a complete discussion on beta distributions. Figure 1 sketches the shape of the beta distribution curves for some values of the beta parameters α and β. We assume the distributions to be statistically independent. This is based on the Kleinrock independence approximation [10]. Basically, under this assumption, the random distributions of the timing parameters become independent even

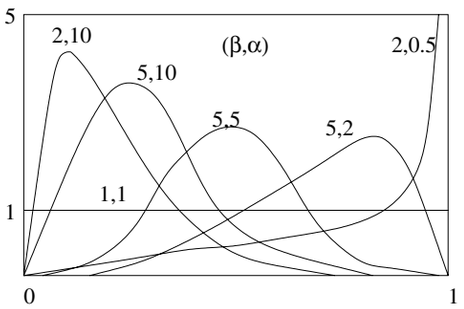


Fig. 1. Shapes of beta distribution for some values of the beta parameters, β and α

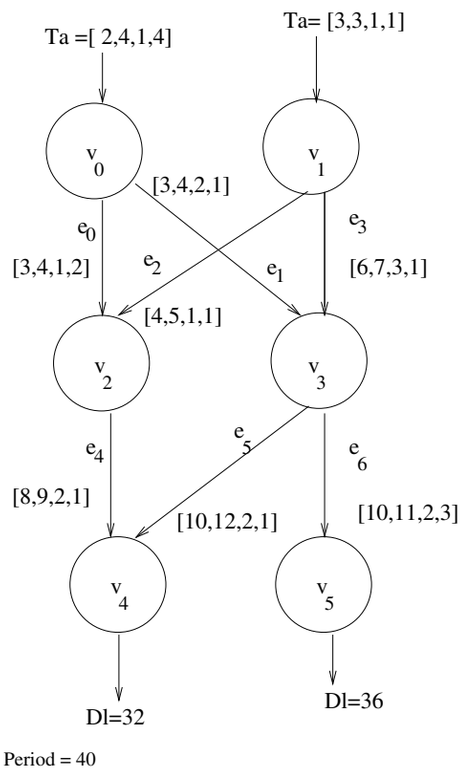


Fig. 2. Example Task Precedence Graph input to the co-synthesis algorithm , $\mathbf{TG_6}$ with $\hat{Pr}$= 40

if they were initially correlated. Factors such as merging of data streams from different sources at the nodes of a task graph, and queuing up of tasks for availability of shareable resources, effectively nullify such correlations. An example of a task graph is illustrated in Figure 2. We consider here only periodic task graphs with a period $\hat{Pr}$. The task invocations and executions are invoked once in every period.

In the following, random variables are denoted in capital letters and sets in bold capital letters. The vertices **V** of **TG** represent the computational tasks of the application and the directed edges **E** represent the data transfer paths or communication tasks. Each task inherits the period P of the task graph. The weight of an edge, $DV(d)$, is the volume of data flowing along the edge e_d. This has a beta distribution; the tuples representing the weights in Figure 2 consist of: lower limit Ll, upper limit Hl, the beta distribution parameters, α and β. The time required for transferring the data along an edge is its weight divided by the data transfer rate of the bus to which it is allocated. The primary inputs to **TG** is the set **I**. Each input has a time of application w.r.t. the beginning of a period. This is a random variable since it can be produced as an output of another system. The primary outputs of the system is the set **O**. Each output is associated with a fixed deadline.

Table 1. (a) Task Execution Times (b) Processor Database (c) Bus Database

Task	P1	P2	P3	P4
v_0	2,2,1,1	2,2,1,1	*	*
v_1	4,5,3,1	4,5,1,2	*	*
v_2	10,12,4,2	8,10,4,2	*	2,3,5,1
v_3	15,20,2,1	13,18,2,2	4,6,1,1	*
v_4	8,12,3,1	4,5,5,5	*	*
v_5	5,7,4,1	8,10,1,2	1,3,1,1	*

(a)

Property	P1	P2	P3	P4
Cost	1000	2000	5000	2500
Busses Supported	B1 B2	B1 B2 B3	B1 B3	B2

(b)

Property	B1	B2	B3
Cost	200	900	2000
Transfer Rate	10	8	80

(c)

Each task can be implemented on a number of processors as shown in Table 1-a. Each cell in this table represents a beta distribution. A cell containing '*' indicates that the task in its row cannot be implemented on the processor in its column. The technical specification of the set of available processors **P** and busses **B** is stored in a database as shown in Table 1-c and Table 1-d respectively. When two communicating tasks are mapped to the same processor, the data transfer is assumed to take place through local memory and the associated communication time is considered negligible.

In the ensuing discussion, the computation and communication tasks are treated uniformly. Let **VE** = **V** U **E**. The execution time of a computation task and the data transfer time of a communication task are referred to commonly as 'processing time'. Processors and busses are treated uniformly as a set of resources **R**.

The task graph is associated with the following timing parameters:

1. $TA(x)$: The time of application of primary input i_x .
2. $Dl(y)$: The fixed deadline at primary output o_y.
   Each task ve_k has the following random timing variables:
3. $TES(k)$: The earliest time when it can start execution.
4. $TPR(k, Type(j))$: The processing time of the task on the type of the resource r_j .
5. $TEC(k)$: The earliest time when the task can complete. This is equal to its earliest start time plus its processing time.
6. $TLC(k)$: The latest time by which the task must be completed, so that the system deadlines are met.
7. $TR(k)$: The time at which all the input data required for executing the task ve_k becomes available.
8. $TS(k)$: The time when the task actually starts executing.
9. $TC(k)$: The time at which the task is completed. If the task is allocated to r_j, then $TC(k) = TS(k) + TPR(k, Type(j))$.

When two tasks are cascaded in series, their execution times are added to obtain the final distribution. From the theory of probability , we know that the probability density of a random variable Z , pdf_Z(z) when Z = X+Y, is given by the convolution of the density functions of X and Y. Assuming the density functions to be zero for negative values of the random variables,

$$pdf_Z(z) = \int_0^{+\infty} pdf_X(x) pdf_Y(z-x) dx \tag{1}$$

When a vertex has a fan-in > 1, its earliest start time is the maximum of the finish times of all its fan-in edges. The distribution of a random variable, Z which is the maximum of two random variables X and Y is given by:

$$pdf_Z(z) = pdf_X(z) cdf_Y(z) + pdf_Y(z) cdf_X(z) \tag{2}$$

where cdf() is the cumulative distribution function.

Given a deadline for a task to be completed, the latest time when it should start is the specified deadline minus its processing time . The probability density of the difference $Z = X - Y$ is given by:

$$pdf_Z(z) = \int_0^{+\infty} pdf_Y(x) pdf_X(x - z) dx \tag{3}$$

If a vertex has a fanout > 1, its latest time for completion is equal to the minimum of the distributions of the start times of all its fanout edges. If Z = minimum (X,Y) its density function is given by;

$$pdf_Z(z) = pdf_X(z)(1 - cdf_Y(z)) + pdf_Y(z)(1 - cdf_X(z)) \tag{4}$$

The penalty function: There is a penalty associated with scheduling each task ve_k.If the scheduler assigns it a release time which results in its completion time exceeding its latest time for completion, then a positive penalty is incurred. The penalty function is,

$$f_k(TC_k) = ExpVal(TC_k - TLC_k) \tag{5}$$

For any feasible schedule, the penalty functions should be negative. Given a shared resource to which two or more tasks have been mapped, an optimum uniprocessor schedule is one which minimizes the maximum penalty function amongst all these tasks [2].

3 The Probabilistic Scheduling Algorithm

The topological characteristics of the task graph itself imposes a partial ordering of the release time for its tasks. If $ve_k \longrightarrow ve_l$, it indicates that ve_k must complete before ve_l can begin. The task graph is initially partitioned into levels $\mathbf{L}$ such that level $\mathbf{L_m}$ consists of the immediate successors of the tasks (vertices or edges) in $\mathbf{L_{m-1}}$. The lowest level $\mathbf{L_0}$ consists of those tasks which receive primary inputs only. The genetic algorithm described in the next section, generates randomized feasible resource selections and task allocations for a population of chromosomes. As a result of the allocations shared-resource groups are created. A group is denoted as $\mathbf{G(j, \bar{V}E)}$, where each $ve_k \in \bar{VE}$ is mapped to the same resource r_j. Each chromosome is subject to a uniprocessor non-preemptive scheduling algorithm which aims to minimize the maximum penalty function associated with each resource. The algorithm described below is outlined in Figure 3.

Uniprocessor scheduling: Scheduling starts from level L_0 and progresses towards higher levels successively. The tasks in a level $\mathbf{L_m}$ are first partitioned into shared-resource subgroups, $\mathbf{G_m(j, \hat{V}E)}$, such that (i) each $ve_k \in \mathbf{\hat{V}E}$ is allocated to the common resource r_j (ii) $\mathbf{\hat{V}E} \subseteq L_m$. It can be seen that $\mathbf{G(j, \bar{V}E)}$

$\supseteq \{\mathbf{G_m}(\mathbf{j}, \hat{\mathbf{VE}})\}$. For each subgroup, the scheduler needs to determine the order and time for releasing the tasks in it. It is evident that the scheduling of different subgroups within the same level are mutually independent.

Latest completion times: First, the distribution of the latest time within which each task in **TG** must be completed is evaluated. Let **FO**(**k**) be the set of fan-out paths emanating from task ve_k. The delay of the path $fo(k)_z \in \mathbf{FO}(\mathbf{k})$ is,

$$TOD(k)_z = \sum_{ve_a \in fo(k)_z} TPR(a, Type(j)) \tag{6}$$

The distribution of $TOD(k)_z$ is evaluated by applying Equation 1 to the tasks along the path $fo(k)$. Let the primary output terminating this path be o_y, with a deadline $Dl(y)$. The latest time by which the task ve_k must be completed is given by;

$$TLC(k) = \min_z \{Dl(y) - TOD(k)_z | o_y \in fo(k)_z\} \tag{7}$$

The distribution of $TLC(k)$ is evaluated using Equation 4. The latest completion time distributions of each task is evaluated by applying Equations 3 and 4 recursively, starting with the output tasks.

Earliest start times: The earliest time when a task ve_k can start is determined by its precedence constraints as given by the topology of **TG** and the resource sharing constraint (the resource assigned for a task may be occupied by other tasks not having any precedenc relationship with it, even after all its predecessors have completed).

(i) Precedence constraints The release time of a root level task is equal to the time of application of its input. For a task ve_k^m in any other level L_m, the release time can be evaluated as,

$$TR(ve_k^m) = \max_j \{TC(ve_j^{m-1})\}$$

$$\forall j : ve_j^{m-1} \longrightarrow ve_k^m \tag{8}$$

The *pdf*(.) of the release time is evaluated by repeated application of Equation 2. The earliest start time of task ve_k is initially set equal to its release time.

$$TES(k) = TR(k) \tag{9}$$

If ve_k has been mapped to a resource exclusively, then its earliest start time is fixed up as given by Equation 9. Otherwise, the waiting time on the shared resource must be added.

(ii) Resource sharing constraint
Let the ordered set $\tilde{\mathbf{VE}}$ contain tasks that have been already been scheduled on r_j, arranged in increasing order of their release times. The algorithm searches the first task in this set, ve_l, whose upper limit of completion time distribution, $Ul(TC(l))$ is greater than the lower limit of start time of the candidate task to be scheduled, $TES(k)$. This gives the first position for a sliding window to scan through the processor occupancy for a free time slot to execute ve_k. The distribution of the free time ΔT between $TES(k)$ and $TS(l)$ is,

$$pdf(\Delta T) = pdf(TS(l) - TES(k)) \tag{10}$$

The lower and upper limits of $pdf(\Delta T)$ gives the minimum and maximum free time available for executing ve_k on r_j, just before ve_l. The probability that this free time is greater than the processing time of ve_k on r_j is:
$Pb_{free}(k, l, j) =$

$$\int_0^{+\infty} pdf_{TPR(k,Type(j))}(t)(1 - cdf_{\Delta T}(t))dt \tag{11}$$

The Risk parameter, τ: We define an input global risk parameter $0 \leq \tau \leq 1$ which gives a decision threshold for determining whether the free time slot is enough for the candidate task. If $Pb_{free}(k, l, j) \geq \tau$,then the available time slot ΔT is considered suitable for fitting in ve_k. Otherwise, the next attempt to lodge it is made between ve_l and ve_{l+1}. In the latter case, the distribution of $TES(k)$ must be modified because ve_k cannot start before the completion of ve_l:

$$pdf(TES(k)) = pdf(\max(TES(k), TC(l))) \tag{12}$$

Note that higher the preset value of τ, lower that is risk that when a task is scheduled within a free slot on a resource, its processing time overshoots the available free time. The search process continues till a free time slot that can accommodate ve_k is found.

Release times: The earliest completion time of the task ve_k is,

$$TEC(k) = TES(k) + TPR(k, Type(j)) \tag{13}$$

The priority Pry(k) assigned to the task ve_k is set equal to its penalty function at the earliest completion time, viz., $f_k(TEC_k)$.

$$Pry(k) = f_k(TEC_k) =$$

$$\mathbf{ExpVal}[pdf(TEC(k) - TLC(k))] \tag{14}$$

The task can meet its deadline only if this quantity is negative. A higher priority value means that the task has a tighter relative deadline . The tasks are scheduled in decreasing order of their priorities.

As the tasks in a resource group are released, the free slot availability pattern of the shared resource changes. In order to evaluate the release time , each task , barring that with the highest priority, is subject to the search process once again, so that it is placed in a free slot. This step determines the release and completion times of all the tasks in the resource group. After all resource groups in a level are scheduled, tasks with exclusive mappings and/or local communication tasks are scheduled trivially.

Task re-scheduling: Consider the effect of a situation where $\tau < 1$ and $Pb_{free}(k, l, j) \geq \tau$ but less than unity. This condition implies that there is a finite probability that the execution time of ve_k exceeds the free time available on r_j just before the release of ve_l. Since ve_l cannot start before the completion of ve_k, its release time distribution must be modified as,

$$pdf(TS(l)) = pdf(\max(TS(l), TC(k))) \tag{15}$$

The re-scheduling of a task which belongs to a level lower than the one currently being scheduled, would obviously have a ripple effect on all its successors which have been scheduled so far. Each task re-scheduled may also affect the release times of subsequent tasks in their respective resources, if there in an overlap in execution times . Thus a domino like chain reaction is initiated until all tasks affected up until the current level are re-adjusted.

Performance evaluation: After all levels have been scheduled, the completion time distributions at the primary outputs become known. The completion time of output o_y is equal to the completion time of the task which generates it. The probability of meeting the deadline specified for an output o_y is given by,

$$Pb_{dm}(y) = \int_0^{Dl(y)} pdf_{TC(y)}(t)dt \tag{16}$$

The performance fitness or timeliness is defined as the probability that all specified deadlines are met . Denoting this as Fp , we have;

$$Fp = \prod_y Pb_{dm}(y) \tag{17}$$

The overall complexity of the scheduling algorithm arises from two factors: the maximum range of the distributions, g, and the total number of tasks in the task graph, n. The time taken for evaluating the latest completion times and the earliest start times taking into account the topological constraints only, is $O(n^2 + ng^2)$. Let m be the number of tasks in a shared-resource sub-group belonging to a level. The total time taken for modifying the earliest start time of a task w.r.t. resource sharing constraints and re-scheduling tasks affected by execution time overlaps is bounded by $O(nwg^2 + n^2g^2)$.

procedure STOC-SCHED(**TG**, chromosome: c, τ)
begin
For the allocations given by c:

1. Initialize : Calculate latest completion times distributions of each task, $TLC(k)$: $ve_k \in \mathbf{TG}$
2. Schedule:
   repeat for each level $\mathbf{L_m}$ of the task graph,
   2.1 Group together tasks $\in \mathbf{L_m}$ which share the same resource r_j.
   2.2 **for** each group $\mathbf{G_m(r_j, \hat{VE})}$
   2.2.1 **for** each task $ve_k \in \mathbf{\hat{VE}}$
   a. Evaluate the distributions of the earliest start time , $TES(k)$ w.r.t. precedence constraints .
   b. Modify the distributions $TES(k)$ w.r.t free time slot availability in shared resources using the guidance parameter τ . Evaluate the distribution of the earliest completion time $TEC(k) = TES(k) + TPR(k, Type(j))$.
   c. Calculate deadline priority: $Pry(k) = \mathbf{Exp}(TEC(k) - TLC(k))$
   end for
   2.2.2 Sort tasks in the group w.r.t. their priorities
   2.2.3 Release the tasks in decreasing order of priority
   • **for** each task in the sorted group
   a. Evaluate the distributions of the release time $TS(k)$ and Completion time $TC(k)$ by finding free time slot availability in the shared resource r_j, as per the latest schedule.
   b. Reschedule: If the task completion time $TC(k)$ distribution overlaps the start time distribution of the next task ve_l, reschedule the latter.Reschedule all other affected tasks which have been scheduled so far.
   end for
   2.3 Schedule tasks with exclusive mappings / local communication tasks.
   end for
   until all levels are scheduled

end

Fig. 3. The stochastic scheduling algorithm

4 The Genetic Algorithm

An initial population of chromosomes **C** is set up by randomized feasible allocations and mappings. Each chromosome c_a consists of four parts encoding the following: (i) Pn_a, the maximum number of processors of each type that can be utilized for constructing the computation architecture (ii) Mp_a, the **V** X **P** mappings of computational tasks on processors available (iii) Bn_a the maximum number of each type of bus available, and (iv) Mb_a the **E** X **B** mappings of the data transfer paths on the available busses .

Each chromosome is subject to the scheduling process as described in section 2, which gives its performance fitness F_p. The cost of the architecture Ct is the sum of the costs of the processors and busses actually used for implementation. If Ct_{max} is the maximum cost allocated for the project, the cost fitness is defined as :

$$Fc = \frac{Ct_{max} - Ct}{Ct_{max}} \tag{18}$$

The overall fitness Fo is :

$$Fo = Fp \times Fc \tag{19}$$

The chromosomes are sorted in decreasing order of their overall fitness values. The mutation, crossover and selection operators have been used for evolution. Convergence is achieved when the best fitness remains the same over a number of successive generations . We adopt a hierarchical approach for evolving the initial population. In the first phase, the maximum number of each type of processor required is determined. In the second phase, the **V** X **P** mappings of each chromosome is optimized separately, by evolving a clone population for each parent chromosome ,bearing the same Pn part but with the other parts regenerated. The best clone evolved replaces the parent. At the end of this phase, a set of optimized computation task allocations is obtained. Each computation architecture has its own communication requirements. In the third and fourth phases the maximum number of each type of bus required, and the **E** X **B** mappings are evolved respectively, in a manner analogous to the first and second stages. During the **E** X **B** evolution phase, the clones for each parent have replicated P_n, M_p and B_n parts , but the M_b parts are regenerated. At the end of the fourth phase, the communication task allocations are optimized.

5 Results

The implementation of the software is in about 18000 lines of C++ code on PentiumIII(450MHz) PC over a Linux platform.

Experiment 1: **Best architecture**

We tested the algorithm using the task graph $\mathbf{TG_6}$ in Figure 2 and the inputs shown in Table 1 . Table 2 shows the processors and busses selected and the task allocations for the two best chromosomes along with their fitness values.

Note that the best solution uses one general purpose processor and one special purpose processor which gives a high performance execution of tasks v_3 and v_5, whereas the second best solution uses four GPPs. The best solution gives very high performance, but the second solution has a better cost fitness with only a slight degradation of performance. Figure 4 shows the completion time distributions at the two primary outputs v_4 and v_5 for the best chromosome.

Table 2. The architectures represented by two best chromosomes and their fitness values

Chromo. 1		**Chromo. 2**	
Type	**Tasks**	**Type**	**Tasks**
1. P2	v_0,v_1 v_2,v_4	1. P2	v_3,v_0
		2. P1	v_1,v_2
2. P1	v_3,v_5	3. P2	v_4
		4. P1	v_5
1. B1	e_1,e_3 e_5	1. B2	$e0$,$e4$ $e5$
FITS:		2. B1	e_3,e_6
Perf.	0.9999	Perf.	0.9202
Cost	0.8475	Cost	0.8534
Overall	0.8475	Overall	0.7854

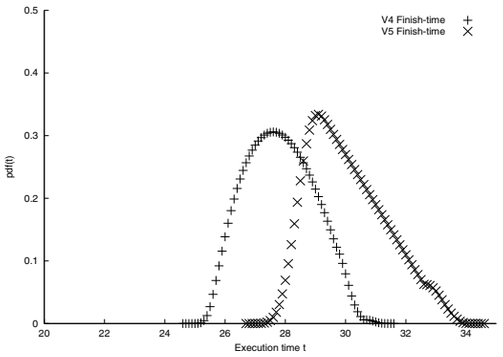


Fig. 4. The completion time distributions at the primary outputs of task graph $\mathbf{TG_6}$ (figure 2) for the best chromosome

Experiment 2: **Deterministic versus random execution times**
In this set , we used two sets of input data. In one set, fixed values for the task

execution times were used, and in the other set random distributions were used. The deterministic timings were set equal to the upper limits for the corresponding distributions. Note that in the deterministic case, only binary values of the performance fitness possible (all deadlines met or at least one deadline not met). The results obtained are tabulated in Table 3. In the first case, the deadlines were set to be slightly less than the critical path delay to each output (considering both precedence and resource sharing constraints). The deterministic case did not give any viable solution, whereas in the random case, the best solution gave a performance fitness of 0.9686; quite acceptable for a soft real time system.

Table 3. Results of co-synthesis for the deterministic and stochastic cases(a) Tight deadlines (b) Relaxed deadlines

Deadlines	Execution Timings	Performance Fitnesss	Cost Fitness	Overall Fitness
case a: **Tight**				
$Dl_4 = 30$	Deterministic	0	*	0
$Dl_5 = 35$	Stochastic	0.968	0.834	0.807
case b: **Relaxed**				
$Dl_4 = 37$	Deterministic	1	0.867	0.867
$Dl_5 = 37$	Stochastic	1	0.923	0.923

In the second case, the deadlines were relaxed. Both models produced viable solutions, each giving a performance fitness of 1. However, the solution provided by the deterministic model is costlier as reflected by its lower cost-fitness of 0.8673 as against 0.9227 for the stochastic model. Figures 5a and 5b show the respective task-occupancy of all the processors used in the two architectures. Note that only two processors are used in the architecture produced for the case of random execution times, and they are utilized optimally , whereas the four processors used in the architecture produced for the case of deterministic execution times are underutilized.

The Utilization of a processor is defined as the ratio of its busy time to the sum of its busy and free times within the duration of one period. Giving 50% weightage to the probabilistic timings in Figure 5a (*i.e.* the unfilled and lightly shaded portions), and considering the overlapping distributions on a processor only once, the average utilization for all processors used in the stochastic and deterministic cases turn out to be 51.25% and 35.83% respectively. Note that the performance fitness is unity in both cases, as is the requirement for *hard* real time systems. Therefore , even when such systems are targeted, the stochastic approach yields a superior solution in terms of cost and processor utilization.

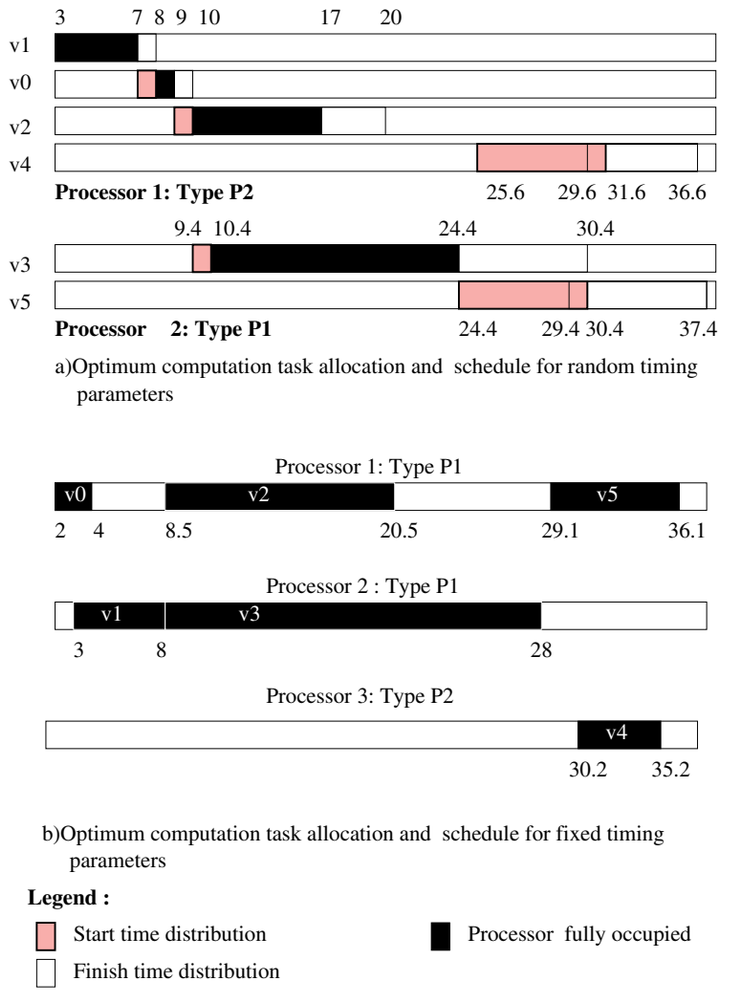


Fig. 5. Processor utilizations for (a)Stochastic model (b) Deterministic model

Experiment 3: **Effect of the beta parameters on performance fitness**
We applied the co-synthesis algorithm to the task graph in Figure 2 repeatedly, each time varying the beta parameters of the execution time distributions of v_3, but keeping all other distributions the same . This task falls into the critical paths of both the primary outputs. The results illustrated in Figure 6 show that the performance fitness is directly proportional to α and inversely proportional to β. This is easily understood when we examine the shapes of the curves in Figure 1. The expected values of the distributions decrease when α is increased, and increase when β is increased.

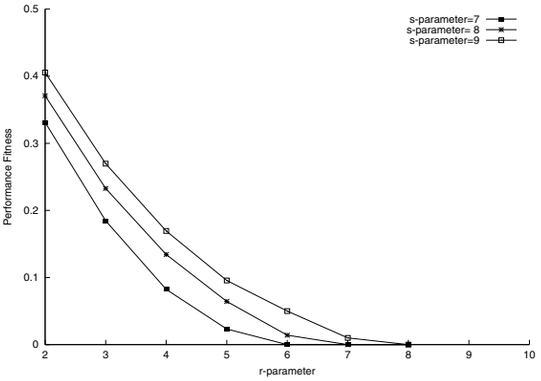


Fig. 6. Effect of the beta parameters α and β on performance

Experiment 4: **Results of GA based optimization**
The variation of overall fitness with successive generations is shown in Figure 7. Note that during the second phase, which is responsible for optimizing computation task allocations through 'clones evolution', the algorithm approaches the best fitness value rapidly. Further marked improvement is brought about during the last phase which optimizes communication task allocations. The optimum architecture is frozen within 80-90 generations.

We next applied the co-synthesis algorithm to a number of task graphs. Figure 8 shows the performance fitness versus the cost fitness of the chromosomes in the final population, for three representative task graphs. In some cases, typified by curve 'a', the evolution process produced a single best solution, given by point 1 on the curve . However, for many task graphs, the final results were a set of solutions with almost the same overall fitness values. They offer a range of choices in terms of cost-performance tradeoff. Curve 'b' represents such an outcome. Some experiments produced a subset of the final solutions offering exchangeability between cost and performance ,like the solutions corresponding to the knee points 2 and 3 on the curve 'c'.

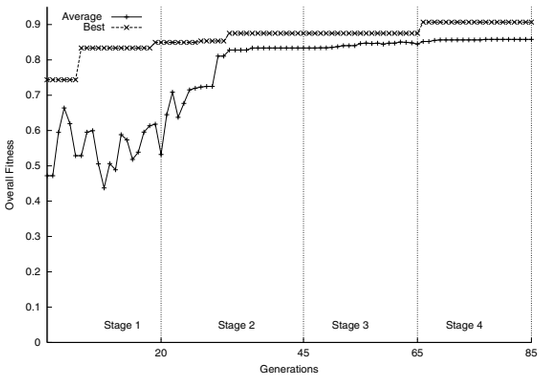


Fig. 7. Overall Fitness versus Generation during the evolution process for the inputs given in Figure 2.

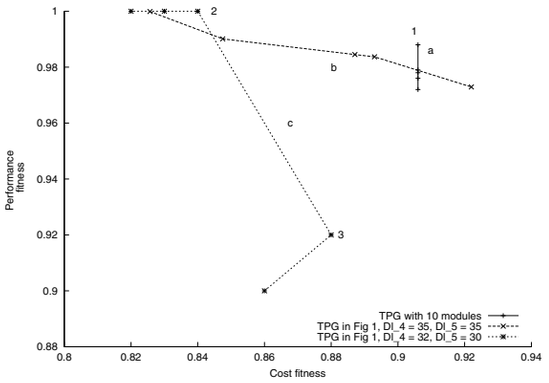


Fig. 8. Performance fitness versus cost fitness of the solutions obtained for three different task graphs

Experiment 5: **Performance of the stochastic co-synthesis algorithm**
Table 4 shows the CPU run times taken when the algorithm was applied to a set of task graphs. The period of the task graphs ranged from 40 to 100 seconds. The beta parameters, α and β of the execution time distributions of different tasks ranged from 1 to 9 (including fractional values). The execution time increases with the number of tasks. It also increases rapidly with the maximum range of execution time distributions. This is expected because of the compute-intensive nature of the random variable functions involving floating point matrix operations.

Table 4. CPU run times for the stochastic scheduling algorithm

Graph	Max. Range	Nodes	Edges	Run Time
TG_5	5	5	5	0.08
TG_{6a}	5	6	6	0.11
TG_{10}	5	10	12	0.35
TG_{12}	5	12	15	0.69
TG_{16}	5	16	21	1.25
TG_{18}	5	18	23	2.46
TG_{20}	5	20	28	5.37
TG_{6a}	5	6	6	0.11
TG_{6b}	10	6	6	0.40
TG_{6c}	20	6	6	0.75

6 Conclusions

We have proposed a stochastic model based on which we derive a method for task scheduling in a heterogeneous multiprocessor architecture for a given task graph, and evaluating its performance. We have applied a mixed approach, using a genetic algorithm whereby the resources' selection and task allocations are encoded into the chromosome structure, and the stochastic scheduling algorithm is embedded into the fitness evaluation step. To the authors' knowledge, the stochastic approach has been applied for the first time in the context of co-synthesis in this work. The random distributions of communication times have also been taken into account in our model. The superiority of this approach over the deterministic model has been clearly demonstrated for the design of both hard and soft real time applications. We have shown that the phased and hierarchical adaptation of genetic algorithms to the co-synthesis problem results in very fast convergence. It generates a range of optimized solutions for the given real time application, giving alternatives for cost performance tradeoff.

References

1. J.K. Antonio and Y.A. Li. Estimating the execution time distribution for a task graph in a heterogeneous computing system. In *Proceedings of Heterogeneous Computing Workshop*, pages 172–184, April 1997.
2. K.R. Baker, A.H.G.R. Kan, E.L. Lawler and J.K. Lenstra. Preemptive scheduling of a single machine to minimize maximum cost subject to release dates. *Operations Research*, pages 381–386, Mar-Apr 1983.
3. T. Benner, R. Ernst and J. Henkel. Hardware-software cosynthesis for microcontrollers. *IEEE Design and Test of Computers*, 10(4):64–75, Dec. 1993.
4. B.P. Dave and N.K. Jha. Cohra:hardware software co-synthesis of hierarchical distributed embedded system architectures. *IEEE Transactions on Computers*, pages 347–354, Oct. 1997.
5. B.P. Dave and N.K. Jha. Cofta:hardware software co-synthesis of heterogeneous distributed embedded systems for low overhead fault tolerance. *IEEE Transactions on Computers*, 48(4):417–441, April 1999.
6. C. Derman, L.J. Glesser and I. Olkin. *Probability Models and Applications*. Macmillan, 1994.
7. A.K. Gupta and C.P. Ravikumar. Genetic algorithm for mapping tasks onto reconfigurable parallel processors. *IEE proceedings of Computer Digital Technology*, 142:81–86, March 1995.
8. C.J. Hou and K.G. Shin. Allocation of periodic task modules with precedence and deadline constraints in distributed real time systems. *IEEE transactions on computers*, 46(12):1338–1355, December 1997.
9. A. Kalavade and E.A. Lee. A hardware-software codesign methodology for dsp applications. *IEEE Design and Test of Coputers*, 10(3):16–28, Sept. 1993.
10. L. Kleinrock. *Communication Nets: Stochastic Message Flow and Delay*. McGrawHill, NY, 1964.
11. C.M. Krishan and K.G. Shin. *Real-Time Systems*. McGraw-Hill, 1997.
12. V. Nag. Synthesis of fault tolerant heterogeneous multiprocessor systems. Master's thesis, Dept. of Electrical Engineering,Indian Institute of Technology, New Delhi, April 1997.
13. A. Parker and S. Prakash . Sos:synthesis of application specific heterogeneous multiprocessor systems. *Journal of Parallel and Distributed Comput.*, 16:338–351, Dec 1992.
14. H.J. Siegel and M. Tan. A stochastic model of a dedicated heterogeneous computing system for establishing a greedy approach to developing data relocation heuristics. In *Proceedings of Heterogeneous Computing Workshop*, pages 122–133, April 1997.

A Fault Tolerance Extension to the Embedded CORBA for the CAN Bus Systems

Gwangil Jeon[1], Tae-Hyung Kim[2], Seongsoo Hong[3], and Sunil Kim[4]

[1] Seoul National University, School of Computer Science and Engineering, Seoul 151-742, Korea
gijeon@ssrnet.snu.ac.kr
[2] Hanyang University, Department of Computer Science and Engineering, Kyunggi-Do 425-791, Korea
tkim@cse.hanyang.ac.kr
[3] Seoul National University, School of Electrical Engineering, Seoul 151-742, Korea
sshong@redwood.snu.ac.kr
[4] Hongik University, Department of Computer Engineering, Seoul 121-791, Korea
sikim@cs.hongik.ac.kr

Abstract. This paper presents a fault tolerant extension to our CAN-CORBA design. The CAN-CORBA is an environment specific CORBA we designed for distributed embedded control systems built on the CAN bus. We extend it for fault tolerance by adopting passive and active replication strategies mandated by the OMG fault tolerant CORBA draft standard. To reduce resource demands of these fault tolerance features, we adopt a state-less passive replication policy and show that it is sufficient for embedded real-time control applications. We give an example CORBA program with its IDL definition to demonstrate the utility of our fault tolerant CAN-CORBA. The newly extended CAN-CORBA clearly reveals that it is feasible to use fault tolerant CORBA in developing distributed embedded systems on real-time networks with severe resource limitations.

1 Introduction

Many embedded system applications require a high degree of fault tolerance for their responsive and continuous operations since they often run in a harsh environment and possess stringent timing constraints. Unfortunately, writing programs for embedded systems, even without fault tolerant features, is already a seriously complicated task. Topping off fault tolerance requirements easily lead embedded system developers to the infamous embedded software crisis.

Recently, in [9] and [10], we proposed a new embedded CORBA design dedicated for CAN-based distributed control systems as an effort to provide a solution to the complexity problem of embedded software systems. We named this new CORBA design *CAN-CORBA* and demonstrated that it would be feasible to

J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 114–133, 2001.

use it in developing distributed embedded systems on real-time networks. There were two major difficulties we faced during the design of CAN-CORBA. First, the resource demands of the original CORBA implementation easily exceeded the 1 Mbps network bandwidth of the CAN bus. This required many optimizations in the transport protocol and inter-ORB protocol design for the new CORBA. Second, there was a crucial distinction in communication models between the conventional CORBA and typical control systems: connection-oriented vs. group communications. Our CAN-CORBA was designed to solve both problems.

In this paper, we extend our original design of CAN-CORBA for fault tolerance since it is one of inevitable requirements imposed on mission critical embedded real-time systems. Recently, the OMG (Object Management Group) made a request for a proposal to define a specification of fault tolerant CORBA [4]. At the time of submitting this paper, it was in the process of voting to adopt a standard fault tolerant CORBA after it had received the joint revised submission in December 1999 [6]. This submission was finally approved by the board of directors in March 2000.

As mandated by the draft specification, our fault tolerant schemes are based on object replication. For our CAN-CORBA, we adopt two replication strategies mandated by the draft fault tolerant CORBA specification. These are passive replication and active replication. Since the straightforward application of these strategies leads to excessive resource demands in embedded real-time systems, we propose a passive replication policy that does not require message logging and object state transfers. We show that such a state-less passive replication strategy is sufficient for applications running on top of our CAN-CORBA. We also show that active replication will be implemented in a straightforward manner in our CAN-CORBA due to the reliable broadcast bus of the CAN. To demonstrate the utility of our fault tolerance schemes, we give an example CORBA program with an IDL definition.

The remainder of the paper is organized as follows. Section 2 gives the target system hardware model and the publisher/subscriber communication schemes that our replication mechanisms are based on. In Section 3 we introduce the general replication strategies for fault tolerance and then present what are peculiar to the CAN-based applications and the CAN-CORBA transport protocols. Based on the identified peculiarities, we present the various replication mechanisms under the conjoiner-based CAN-CORBA transport protocols in Section 4. Section 5 presents an example program that demonstrates the usage of our extended transport protocol that includes fault tolerance in CAN-CORBA. Finally, Section 6 concludes this paper.

1.1 Related Work

Two issues are essential in designing a fault tolerant system: fault detection and replication management. Rajkumar and Gagliardi proposed a publisher/subscriber model for distributed real-time systems in [13] and developed a fault-tolerant extension to their publisher/subscriber model in [12]. Specifically, they adopted Cristian's periodic broadcast membership protocol [2] for

fault detection. They assumed that the underlying network was built on top of a point-to-point transport layer. As a result, one-to-many group communication was realized by sending message copies to multiple subscribers via a number of point-to-point connections. This assumption may lead to a performance problem since it is inefficient to simply send periodic "check" messages to detect failures – especially where many publishers and subscribers exist. Fetzer [3] devised an efficient mechanism for fail awareness under the publisher/subscriber communication paradigm. Since CAN provides a reliable broadcast medium, fault detection is hardly an issue in our work. It can be achieved through a simple timeout mechanism.

Synchronizing object states among replicas is one of difficult tasks in implementing a fault tolerant CORBA. Obviously, it is a computationally expensive operation and seriously complicates the resultant system. Unfortunately, the OMG fault tolerant CORBA RFP [4] mandates such a strong consistency and the joint revised submission [6] to the RFP proposes an ORB-level solution by notions of ReplicationManager, PropertyManager, and ObjectGroupManager, to name a few. On the other hand, the minimumCORBA [5] was proposed to remove dynamic facilities for serving requests and for creating, activating, passivating and interrogating objects. In embedded system design practice, decisions on object creation and resource allocation are usually made at design time. Moreover, as will be explicated in Section 3.2, and claimed by [12], the CAN-based embedded application does not require strong consistency among object replicas. Thus, we also excludes unncessary state management and dynamic object management features.

2 System Model

CAN-CORBA is designed to operate on a distributed embedded system built on the CAN bus. In this section, we present the target system hardware model and CAN-CORBA configuration to help readers understand the characteristics and limitations of the underlying system platform, before delving into the details of fault tolerance in our new CAN-CORBA design.

2.1 Target System Hardware Model

We use the same target hardware model as in [9] and [10] since we extend our prior implementation of CAN-CORBA to include fault tolerance. The target hardware consists of a number of function control units (FCU) interconnected by embedded control networks (ECN). As an example of the model, Figure 1 shows the electronic control system of a passenger vehicle. Each FCU, possessing one or more microcontrollers and microprocessors, conducts a dedicated control mission by interfacing sensors and actuators and executing prescribed control algorithms. Depending on configuration, an FCU works as a data producer, a consumer, or both.

As shown in Figure 1, embedded control networks (ECN) connect FCUs through inexpensive bus adaptors. Such ECNs are often required to provide

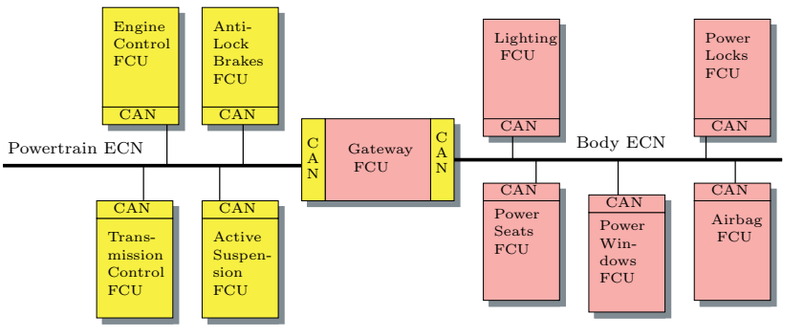


Fig. 1. Example distributed embedded control system: Passenger vehicle control system.

real-time message delivery services and subject to very stringent operational and functional constraints. In this work, we have chosen the CAN [7] as our embedded control network substrate since it is an internationally accepted industrial standard satisfying such constraints.

The CAN standard specifies physical and data link layer protocols in the OSI reference model [1]. It is well suited for real-time communication since it is capable of bounding message transfer latencies via predictable, priority-based bus arbitration. A CAN message is composed of identifier, data, error, acknowledgment, and CRC fields. The identifier field consists of 11 bits in CAN 2.0A or 29 bits in 2.0B and the data field can grow up to eight bytes. When a CAN network adaptor transmits a message, it first transmits the identifier followed by the data. The identifier of a message serves as a priority, and a higher priority message always beats a lower priority one.

The CAN provides a unique addressing scheme, known as subject-based addressing [11]. In the CAN, a message put into the network does not contain its destination address. Instead, it contains a subject tag – a predefined bit pattern in the message identifier which serves as a hint about its data content. A receiver node can program its CAN bus adaptor to accept only a specific subset of messages that carry a specific identifier pattern with them. This filtering mechanism is made possible via a mask register and a set of comparison registers on a CAN interface chip. This subject-based addressing scheme is a key underlying mechanism for the communication models of our CAN-CORBA.

2.2 CAN-CORBA Communication Channels

CAN-CORBA offers a subscription-based, anonymous group communication scheme that is often referred to as "blindcast" or as a publisher/subscriber scheme [13], [8]. In this scheme, a communication session starts when a data producer announces a predefined invocation channel. An invocation channel is a virtual broadcast channel from publishers to a group of subscribers. Data

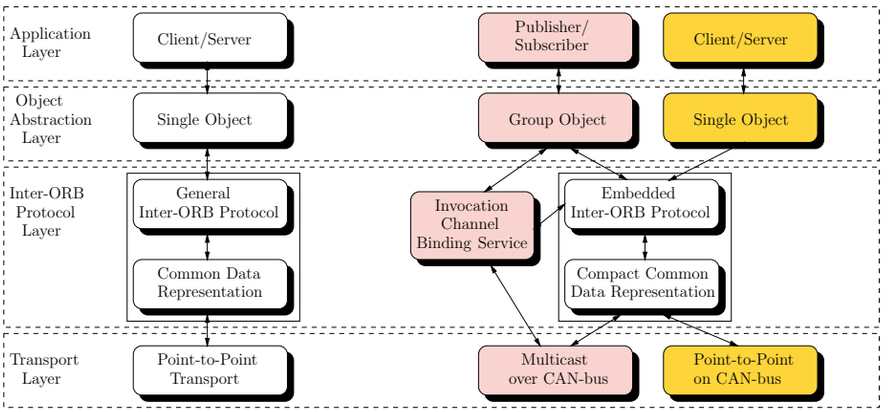


Fig. 2. Comparison between two CORBA configurations.

consumers can subscribe to an announced invocation channel. In this announcement/subscription process, neither a publisher nor a subscriber has to know each other. This anonymity allows for easy reconfiguration of control systems. In CAN-CORBA, an invocation channel is uniquely identified with a CAN identifier, and maintained by the *conjoiner* as described in Section 2.4.

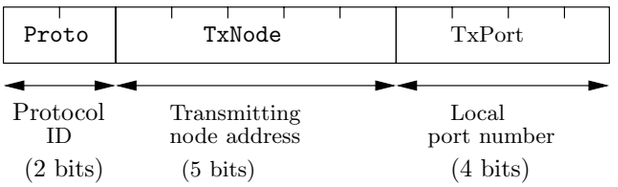
CAN-CORBA also provides point-to-point communication primitives for interoperability with other standard CORBA implementations. Since the CAN bus is a broadcast medium, the publisher/subscriber model is more natural and efficient than the point-to-point model. Moreover, fault tolerance is better serviced by a group communication scheme. Readers are referred to our previous work in [10] for more details on the connection oriented communications in CAN-CORBA.

2.3 CAN-CORBA Configuration

The proposed CAN-based CORBA design stems from the standard CORBA and possesses most of essential components of it. Figure 2 illustrates layer-to-layer comparison between the standard CORBA and the proposed one. Specifically, Figure 2 (b) shows our CAN-CORBA design.

We summarize the noticeable features of our CAN-CORBA.

– **Group object reference:** An object reference in CORBA refers to a single object. It is internally translated into an interoperable object reference (IOR) denoting a communication end-point the object resides on. In CAN-CORBA, an object reference may refer to a group of receiver objects. An intermediary object named a conjoiner is responsible for managing object groups and implementing the internal representation of their references.



Protocol ID	*Usage*
00	reserved
01	EIOP publisher/subscriber
10	EIOP point-to-point
11	binding

Fig. 3. Protocol header format using CAN identifier structure.

- **CAN-based transport protocol:** A new transport protocol is designed to support group communication in CORBA. In this protocol, a sender is totally unaware of its receivers and simply sends out messages via its own communication port.
- **Publisher/subscriber scheme:** A new communication scheme for the publisher/consumer model is also designed on top of the transport protocol. This scheme relies on an abstraction named an invocation channel. It denotes a virtual communication channel which connects a group of communication ports and a group of receivers. Since each port is owned by a publisher, this scheme supports the one-way, many-to-many communication model. In this scheme, a conjoiner object takes care of group management, dynamic channel binding, and address translation. An invocation channel is uniquely identified as a channel tag in an IDL program.
- **Compact common data representation (CCDR):** Common data representation is a syntax which specifies how IDL data types are represented in CORBA messages. In CDR, method invocations often take up tens of bytes in messages. Since a CAN message has only an eight-byte payload, a method invocation may well trigger a large number of CAN message transfers. To deal with this problem, we define the compact CDR. It exploits packed data encoding which avoids byte padding for data alignment, and introduces new data types for variable length integers to encode four-byte integers in a dense form.
- **Embedded inter-ORB protocol (EIOP):** In addition to CCDR, we customize GIOP by simplifying messages types and reducing the size of the IOP headers of messages.

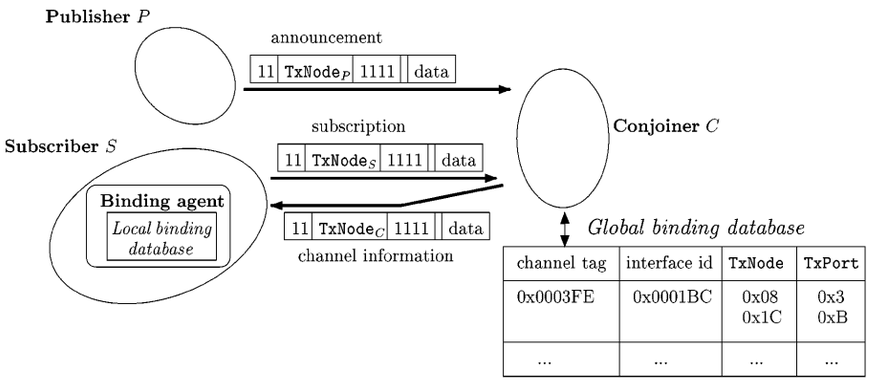


Fig. 4. Conjoiner-based channel binding protocol.

2.4 Channel Binding Protocol for Subscription-Based Communication

Figure 3 shows the protocol header format. We divide the CAN identifier structure into three sub-fields: a protocol ID (`Proto`), a transmitting node address (`TxNode`), and a port number (`TxPort`). They respectively occupy two, five and four bits amounting to 11. The `Proto` field denotes an upper layer protocol identifier. The data field following the identifier in a CAN message is formatted according to the upper layer protocol identifier denoted by `Proto`. In the CAN, a message identifier with a smaller value gets a higher priority during bus arbitration.

The `TxNode` field is the address of the transmitting node. In our design, one can simultaneously connect up to 32 distinguishable nodes with the CAN bus under a given upper layer protocol. The `TxPort` field represents a port number which is local to a particular transmitting node. Since `TxNode` serves as a domain name which is globally identifiable all across the network, `TxNode` and `TxPort` collectively make a global port identifier. This allows ports in distinct nodes to have the same port number and helps increase modularity in software design and maintenance. As the `TxPort` field supports the maximum of 16 local ports on each node, up to 512 global ports coexist in the network under a specific upper layer protocol.

Our channel binding protocol relies on an intermediary object we name a *conjoiner*. It resides on a CAN node whose node identifier is known in advance to every publisher and every subscriber in the system. It must be started right after network initialization and operational during the entire system service period.

The conjoiner maintains a global binding database where each invocation channel has a corresponding entry which is announced and registered by a publisher. Figure 4 illustrates the conjoiner-based publisher/subscriber framework and the global binding database. As shown in the figure, an entry in the global

binding database is a quadruple consisting of a *channel tag*, an *OMG IDL identifier*, `TxNode` and `TxPort`. The channel tag is a unique symbolic name associated with each invocation channel. It is statically defined by programmers when they write the application code. Both publishers and subscribers use it as a search key in the global binding database later on. The OMG IDL interface identifier is a unique identifier associated with each IDL interface in the system. The OMG IDL compiler generates IDL interface identifiers. The CORBA run-time system uses these identifiers to perform type checking upon every method invocation. This ensures strong type safety as required by the CORBA standard. The channel tag and the interface ID together work as a unique name for each invocation channel. It is programmers' responsibility to define a system-wide unique name for an invocation channel.

The conjoiner exchanges messages with a publisher for channel establishment and with a subscriber for channel subscription. When a publisher wants to get attached to an invocation channel, it sends a registration message to the conjoiner. Similarly, a subscriber sends a message to the conjoiner, requesting subscription to an invocation channel. If the conjoiner finds an matching entry for the requested invocation channel in the global binding database, it provides the subscriber with the corresponding binding information. Note that subscribers may be asynchronously informed of changes in its subscribed invocation channel as a publisher is attached to, or detached from the channel. A local binding agent denoted by an oval inside `Subscriber` S node in Figure 4 takes care of such updates.

Note the conjoiner should be able to accept messages from any CAN nodes in the system. Thus, we reserve the local port number (`TxPort`) 1111_2 for this purpose under the network management protocol. As shown in Figure 4, all messages sent to the conjoiner use this local port. Consequently, the conjoiner unconditionally accepts all messages with this port number when `Proto` is 11_2. On the other hand, other CAN nodes can accept messages from the conjoiner in a straightforward way since the `TxNode` and `TxPort` of the conjoiner is known *a priori*.

The data field of a binding message carries full binding information or an actual query. Specifically, a publisher's registration message contains all necessary information to construct a database entry such as a channel tag, an OMG IDL interface ID, and a global port number. Using this information, the conjoiner either creates an entry or modifies one if it exists. A subscriber's request message contains a channel tag and an IDL interface identifier for an invocation channel. On successful retrieval of one or more entries from the binding database, the conjoiner sends a reply message containing information on these entries. These entries are stored into the local binding database of the subscriber.

3 Replication Strategies for Fault Tolerance

In general, fault tolerance is achieved through replication and the basic unit of replication is an individual object. Since non-fault tolerant application is re-

garded as a set of objects posessing single replicas in the program, the definition of fault tolerance may include that of non-fault tolerance, and it has an important subtlety since objects are interoperable with each other whether or not they are replicated.

We begin by introducing two general replication strategies that are mandated by the OMG fault tolerant CORBA draft standard [4], and explicate why stateless passive replication is sufficient for CAN-based applications.

3.1 General Replication Strategies

Replication is made to ensure continuous operation of a particular object. Depending on backing-up styles at the time of fault, replication strategy is classified as *passive* and *active* replication.

In passive replication, only one replica for a given object, which is called a primary replica, executes designated operations, and all others merely wait for an activating signal to be delivered when a fault is detected. According to the creation time of replicas and the length of a message log, it is further classified as cold and warm passive replication.

In *cold* passive replication, non-primary replicas are not created during normal operation, and method invocations and responses are recorded in a message log. When a fault is detected, a recovery service object is initiated and performs a given recovery action using the recorded message log. While this policy imposes no additional memory overhead as long as everything goes well, it requires long recovery time if a fault really occurs.

To shorten the recovery time of cold passive replication, all replicas can be created before faults occur, and the current state of a primary replica is periodically transferred to the others. The state transfer is made through multicast which must be reliable (i.e. not allowed to be lost) and totally ordered with respect to the time of state changes. In the event of a fault, one of the non-primary replicas can be substituted for the faulty primary by recovering only the state since it has been recently transferred. Thus, the recovery time can be reduced. It is called *warm* passive replication.

Note that the length of a message log determines the temperature of passive replication. If it is infinite, a message log should be recorded indefinitely until a fault occurs, thus no need to even create the non-primary replicas until a fault is detected. This is the cold end in the spectrum. If the length is zero, it means that no state is recorded for recovery but immediately transferred to the non-primary replicas via reliable and totally ordered multicast protocol. This is the hot end in the spectrum.

In active replication, when an object invokes a replicated service, all replicas service the request and actively reply with their own results without concerning the faults of other replicas. A client object collects results from all the replicas before making a final decision. Then it can choose one with majority voting, or without majority voting possibly based on the precedence among replicas.

3.2 State-Less Replication Mechanism in CAN-CORBA

In our fault tolerant CAN-CORBA, the state of a primary replica need not be preserved or transferred to non-primary replicas. This allows us to replace the failed primary replica by one of non-primary replicas without transfering the state of the failed primary replica. This argument can be justified in the context of control systems theory. In general, control performance is seriously affected by the freshness of sampled data. Thus, when an embedded control system detects a run-time fault, it is more desirable for the system to start a new sampling period and produce actuation commands using recent data than to attempt to resume the interrupted sampling period using the restored state. As an example, consider a vehicle control system. If an object in an engine control unit does not receive oxygen-level data priodically published by an oxygen sensor, it may use data it received in the previous period. However, if the object cannot retrieve the previous state, possibly because it has been restarted due to a fault, it can re-establish the state using the data that are constantly provided from various publishers. As a matter of fact, it is recovered within a couple of minutes. Fuel efficiency may be lowered temporarily only for the adjusting period.

As a result, it is unnecessary to differentiate cold and warm passive replication for the fault tolerant CAN-CORBA. We thus employ a *state-less* passive replication policy. It is not hot passive since there is no state transfer between replicas during execution. It is not cold passive, either since a faulty primary is not substituted with a stand-by during recovery.

In the next section, we will present the replication strategies that are meaningful to our CAN-based CORBA system, and show the design of the fault tolerant CAN-CORBA using the conjoiner-based publisher/subscriber communication protocol.

4 Replicating CAN-CORBA Objects

As CAN-CORBA adopts the publisher/subscriber communication model for distributed inter process communication, we have three different entities for replication: publishers, subscribers and a conjoiner. Among them, publishers and subscribers are general CORBA objects and the conjoiner is a pseudo CORBA object that was deliberately invented to realize the publisher/subscriber communication model. We present how the replication strategies discussed in the previous section are applied to CORBA objects in our CAN-CORBA. We also show how to distribute and replicate a conjoiner in order to eliminate the single point of failure induced by the conjoiner.

4.1 Passive Replication

Figure 5 illustrates two configurations of a publisher/subscriber connection, each of which respectively denotes a situation before a fault and after its recovery using our state-less passive replication. In the figure, there is one publisher P and

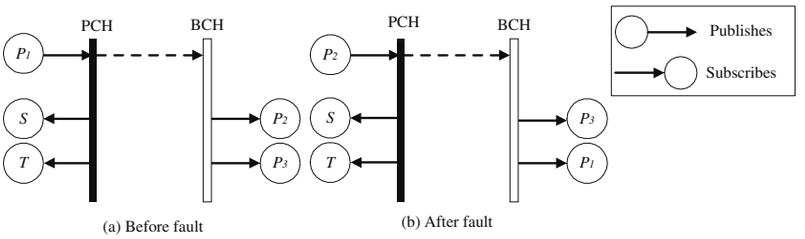


Fig. 5. Passive replication of publisher objects.

two subscribers S and T and their replicas are denoted by numbered subscripts. In our fault tolerant CAN-CORBA design, a publisher/subscriber connection is composed of dual channels: *primary* and *backup* channels. They are denoted by PCH and BCH, respectively. A primary channel (PCH) is used for normal communications while a backup channel (BCH) is used by non-primary publishers to monitor the state of the primary.

The publisher P has three replicas P_1, P_2, and P_3 for the invocation channel PCH where P_1 is the primary replica at the moment. In passive replication, faults are detected using timeouts. The primary replica broadcasts sensor data via PCH and heartbeat messages via BCH. Non-primary replicas do not publish anything and only monitor heartbeat messages. Thus, P_2 and P_3 are attached to BCH as the subscribers of the heartbeat messages. When a non-primary replica detects a fault through timeout, it requests the conjoiner to switch the channels.

The invocation channels after channel switching are shown in Figure 5 (b). Failed publisher P_1 is switched to backup channel BCH as a subscriber to P_2. P_2 becomes a new primary by attaching itself to both PCH and BCH as a publisher. Thus, all entries of {TxNode(P_1), TxPort(P_1)} must be replaced with {TxNode(P_2), TxPort(P_2)} in the global binding table. Becoming a non-primary replica is an autonomous operation performed by a failed primary replica. If P_1's fault is so fatal that it cannot update its local binding database, then it is eliminated from the application.

The following steps illustrate a scenario when P_1 misses the time to publish a heartbeat message.

(1) P_1 announces its registration to PCH and BCH.
(2) S and T request subscription to PCH.
(3) P_2 and P_3 request subscription to BCH.
(4) P_1 periodically publishes messages, S and T keep listening to PCH, and P_2 and P_3 keep monitoring P_1.
(5) P_2 (or P_3 or both) detects a timeout and requests channel switching to the conjoiner.

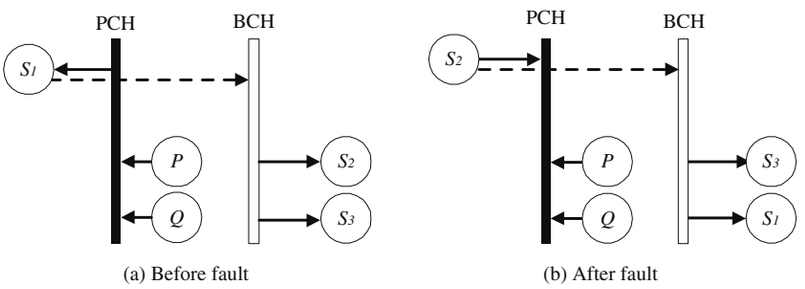


Fig. 6. Passive replication for subscriber objects.

(6) The conjoiner decides the next primary. It broadcasts the newly modified binding information which is a quadruple of (PCH, *primary interface*, TxNode(P_2), TxPort(P_2)). Upon receiving it, objects pertaining to the invocation channel performs the following tasks, respectively.
   - Subscribers S and T update their local binding databases. They have an entry of (PCH, *primary interface*, TxNode(P_2), TxPort(P_2)).
   - P_1 and P_2 switch the roles between primary and non-primary replicas. P_1 updates its local binding database by a quadruple of (BCH, *backup interface*, TxNode(P_2), TxPort(P_2)). If P_1 dies completely, it is discarded. P_1 will not be revived at all.
   - P_3 updates its local binding database. The resulting entry is (BCH, *backup interface*, TxNode(P_2), and TxPort(P_2).

(7) Step (6) completes channel switching. P_2 becomes a new primary and now starts to publish.

As a final note, there are several possibilities for the conjoiner to select the next primary among non-primary replicas. First, the conjoiner chooses in an FCFS fashion. When a fault is detected, many non-primary replicas may send publication requests to the conjoiner. The conjoiner knows the replicated set of publishers *a priori*. It takes the message that comes first and ignores all others. Second, the order is specified by an application programmer, for example, in a circular fashion using replica identifers. More generally, a selection function may be given in an IDL definition by a programmer.

Subscribers are replicated in a similar manner except that the primary subscriber periodically emits an "I am alive" signal to its fellow non-primary replicas. A primary publisher need not send this extra signal since messages periodically published can be used for this purpose. Figure 6 illustrates subscriber replication. The primary subscriber plays a role of a publisher to the non-primary subscribers. When the primary subscriber S_1 is failed and S_2 is substituted for S_1, the whole process is presented as follows.

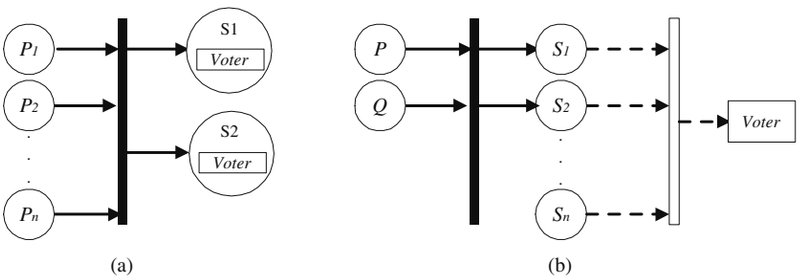


Fig. 7. Active replication for (a) publishers and (b) subscribers.

(1) S_1 requests its subscription to BCH.

(2) P and Q announces their registration to PCH.

(3) S_2 and S_3 request subscription to BCH.

(4) S_1 keeps subscribing to P and Q, sending "I am alive" message via *BCH*, and S_2 and S_3 keep monitoring S_1.

(5) S_2 (or S_3 or both) detects timeout and requests channel switching to the conjoiner.

(6) The conjoiner decides who is a new primary. The conjoiner broadcasts the newly modified binding information which is a quadruple of (BCH, *backup interface*, TxNode(S_2), TxPort(S_2)). The entry of (BCH, *backup interface*, TxNode(S_1), TxPort(S_1)) is eliminated because S_1 is no longer a publisher of "I am alive" message. Upon receiving it, objects pertaining to the invocation channel performs the following tasks respectively.
   - Publishers P and Q are not necessary to be aware of the change.
   - S_1 and S_2 switch the roles. S_1 and S_2 must update their local binding databases by sending queries with {BCH, *primary interface*}, {PCH, *primary interface*}, respectively, in order to switch the channels.
   - S_3 updates its local binding database with an entry of (*BCH*, *backup interface*, TxNode(S_2), TxPort(S_2)).

(7) Step (6) completes channel switching. S_2 is substituted for S_1 and now starts to subscribe.

4.2 Active Replication

In active replication for publishers, a subscriber must subscribe to one and each of replicated publishers, as illustrated in Figure 7 (a) where S and T subscribe to $P_1, \cdots, P_n$. Subscribers have a responsibility to multiplex all data from replicated publishers. A subscriber has freedom to adopt a multiplexing policy. It may use majority voting to tolerate commission faults that are semantically incorrect

data sent by publishers. Or it may serve data on an FCFS basis. In either case, a pertinent voting logic should be included in subscribers.

For subscribers that are actively replicated, an external voter object must be created. Each replicated subscriber publishes its decision to the voter, and the voter finalizes the decision. This is illustrated in Figure 7 (b). N-version programming can be implemented in this way.

4.3 Mixed Replication

It is natural to have mixed replication in an application. For example, it is possible that an actively replicated subscriber listens to a passively replicated publisher.

In our model, "publishers" act as data producers and "subscribers" as data consumers and an object can be a publisher and a consumer at the same time. When an object is replicated, this is done through manipulating invocation channels rather than modifying an entire object module. In a CAN-CORBA application, there are specific code segments that handle invocation channels. Structural changes are made on those code segments to specify replication strategies as will be illustrated in Section 5. Due to the regularity of replicated code structures, an automatic source-level translation scheme can be employed.

4.4 Replicated and Distributed Conjoiner

The conjoiner is an important system resource that takes care of channel binding and services channel switching requests. It also maintains a global binding database. In our original CAN-CORBA design, there is only one instance of a conjoiner, which poses serious reliability and performance problems.

To eliminate the single point of failure introduced by a centralized single conjoiner, the binding database is replicated. As a result, each binding entry is stored at more than two distinct locations. The conjoiner is actively replicated so that any of the replicated conjoiners can deliver correct binding information to its clients. Data consistency among replicated global binding databases is easily maintained using the reliable broadcast CAN bus.

To mitigate performance degradation due to a large number of binding and switching channel requests, the global binding database is distributed or fragmented. To efficiently service publisher announcement requests, each conjoiner replica inserts the binding entry into its database fragment only in its own turn. In this way, the number of entries among distributed conjoiners is balanced. When a subscription request is made, conjoiner replicas need to search only global binding database fragments and thus shorten the response time.

Recall that the presence of the conjoiner is known to all CAN nodes by unique `TxNode` and `TxPort` identifiers. Having multiple conjoiners, each conjoiner replica should have a unique `TxNode` assignment. On the other hand, any two or more conjoiner replicas that share the same disjoint subset of the global binding database should share the `TxPort` as well.

Suppose there are two disjoint sets of the binding database with four replicated conjoiners. The database has two fragments, `db1`, and `db2`. The two pairs of conjoiners (C_1, C_2) and (C_3, C_4) have `db1` and `db2`, respectively. We reserve two sets of local port number 1111_2 and 1110_2 in this case. Roughly speaking, the performance is doubled, and the probability of the conjoiner failure falls fifty percent.

5 Example Programs

```
// IDL
...
interface TemperatureMonitor {
  // Update temperature value for a location.
  oneway void update_temperature(in char locationID, in long temperature);
}

interface FaultMonitor {
  // Publish "I am alive" message.
  oneway void I_am_alive(void);
}
```

Fig. 8. IDL definition for non-primary publisher and subscriber interface.

In this section, we present an example program which demonstrates the usage of our transport protocol extended for the fault tolerant CAN-CORBA. It consists of an IDL interface definition (given in Figure 8), passively replicated publisher code (in Figure 10), and normal subscriber code (in Figure 9). This program denotes the case illustrated in Figure 5. Other combinations of replication such as passively replicated subscribers, and actively replicated publisher/subscriber pairs are similarly written.

The IDL code defines the interfaces of two invocation channels: primary channel (PCH) and backup channel (BCH). `TemperatureMonitor` and `FaultMonitor` interfaces contain the signatures of two methods `update_temperature()` and `I_am_alive()`, respectively. The `update_temperature` method is invoked by a publisher and then executed in a subscriber to update temperature within the subscriber object. It is declared as a `oneway` operation which does not produce output values. Obviously, two-way operation is not allowed in the publisher/subscriber communication protocol.

A primary publisher invokes the `I_am_alive()` method to notify its aliveness to its fellow passive publisher replicas that watch its fault by checking a timeout. Note that programmers may use a periodic message as a heartbeat message to reduce network traffic. In this particular example, an explicit heartbeat message is used, instead since this is more illustrative. Figure 9 and Figure 10 show two

source code files that correspond to a publisher and a subscriber, respectively. Each of the files contains unique channel tag `TEMP_MONITOR_TAG` and an IDL interface identifier `TEMP_MONITOR_IFACE`. Note that `TEMP_MONITOR_TAG` is defined by programmers while `TEMP_MONITOR_IFACE` is generated by our OMG IDL precompiler. Note that `_B_TEMP_MONITOR_TAG` and `FAULT_MONITOR_IFACE` are used only in publisher replicas.

```
// Define a channel tag for temperature monitoring.
#define TEMP_MONITOR_TAG 0x01

// Initialize the object request broker (ORB).
CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv);

// Get a reference to the conjoiner.
Conjoiner_ptr conjoiner =
   Conjoiner::_narrow(orb->resolve_initial_references("Conjoiner"));

// Create a servant implementing a temperature monitor object.
TemperatureMonitor_impl monitor_servant;

// Assign a local CORBA object name to the monitor object.
PortableServer::ObjectId_ptr oid =
    PortableServer::string_to_ObjectId("Monitor1");

// Register the object name and servant to a portable object adaptor
(POA).
poa->activate_object_with_id(oid, &monitor_servant);

// Bind the monitor object to the TEMP_MONITOR_TAG.
conjoiner->subscribe(TEMP_MONITOR_TAG, &monitor_servant);

// Enter the main to receive the temperature values.
orb->run();
```

Fig. 9. Subscriber code (not replicated).

In Figure 9, a subscriber wishing to subscribe to an invocation channel accesses the conjoiner object via `Conjoiner:: subscribe()` method. During this call, the subscriber provides the conjoiner with `TEMP_MONITOR_TAG` and a servant. A servant is a collection of language-specific data and procedures which implement the actual object body. It is written by an application programmer and registered into the CORBA object system via a portable object adaptor (POA). Note that the `TEMP_MONITOR_IFACE` is not explicitly provided during a call to `Conjoiner::subscribe()` method since the `monitor_servant` is a typed object whose interface information can be easily extracted. `Conjoiner::subscribe()` method sends a subscription request message to the conjoiner to get the binding information of an invocation channel. Finally, the subscriber enters into a block-

ing loop where it waits for an invocation of `update_temperature()` method from the publisher.

Figure 10 shows passively replicated publisher code that works for both primary and non-primary publishers. While a primary publisher works only as a data producer, non-primary replicas are in essence heartbeat data consumers that listen to a backup channel. Thus, the given publisher code includes invocations of both channel announcement and subscription. Programmers let the primary publisher register its primary and backup channels via two successive invocations of `Conjoiner::announce()` method. This method allocates a local port from the primary publisher's free port pool and sends an announcement message to the conjoiner. Finally, the primary invokes `update_temperature()` method to broadcast a temperature data message.

Programmers let the non-primary replicas subscribe to the backup channel via the invocation of `Conjoiner::subscribe()` method. During this call, the method provides the conjoiner with the backup channel tag and an object body `ft_detector` that implements the timeout checking logic. Non-primary replicas monitor the primary's status by periodically receiving "I am alive" messages. They wait inside a loop until `ft_monitor->I_am_alive()` is invoked by the primary or until a timeout occurs. If a timeout is detected, they request channel switching to the conjoiner by invoking `Conjoiner::switch()` method. This method returns true if the requesting replica is selected as a new primary. The conjoiner updates the global binding database to replace an old channel binding with a new one. As shown in Figure 3, `TxNode` and `TxPort` of the new channel binding are provided via the protocol header of our transport protocol. Finally, the program resumes publishing temperature data using a different sensor. When the primary channel is switched to the backup channel as a result of the recovery action, some of replicas experience timeouts. In that case, the local binding agent detects the channel switching by examining the local binding database.

6 Conclusions

We have presented a fault tolerant extension to our CAN-CORBA design [9], [10]. The CAN-CORBA is an environment specific CORBA we designed for distributed embedded systems built on the CAN bus. It supports both anonymous publisher/subscriber and point-to-point communications without losing the IDL level compliance to the OMG standard.

In order to support fault tolerance for CAN-CORBA applications, we adopted the OMG fault tolerant CORBA specification and incorporated into the CAN-CORBA both passive and active replication strategies. Since fault tolerance features added excessive complexity to the CAN-CORBA, we took into account the domain characteristics of the CAN-CORBA environment to avoid it. Specifically, we introduced a state-less passive replication policy that did not require message logging and object state transfers. We showed that state-less replication would be sufficient for embedded real-time control applications, as argued

```
// Define a primary channel tag for temperature monitoring.
#define TEMP_MONITOR_TAG 0x01

// Define a backup channel tag for passive replication.
#define _B_TEMP_MONITOR_TAG 0x101

// Initialize the object request broker (ORB).
CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv);

// Get a reference to the conjoiner.
Conjoiner_ptr conjoiner =
    Conjoiner::_narrow(orb->resolve_initial_references("Conjoiner"));

// Obtain references to the temperature (PCH) and
// the replica (BCH) monitor groups.
TemperatureMonitor_ptr monitor =
   conjoiner->announce(TEMP_MONITOR_TAG, TEMP_MONITOR_IFACE);
TemperatureMonitor_ptr ft_monitor =
   conjoiner->announce(_B_TEMP_MONITOR_TAG, FAULT_MONITOR_IFACE);

// Create a servant implementing a fault detector object.
FaultMonitor_impl ft_detector;

// Assign a local CORBA object name to ft_detector.
PortableServer::ObjectId_ptr oid =
   PortableServer::string_to_ObjectId("Ft_detector1");

// Register the object name and ft_detector to a POA.
poa->activate_object_with_id(oid, &ft_detector);

// Bind the ft_detector object to the _B_TEMP_MONITOR_TAG.
conjoiner->subscribe(_B_TEMP_MONITOR_TAG, &ft_detector);

while(1) {

  // Each replica checks if it is a primary one. The conjoiner determines
  // the primary replica by returning TRUE to the ::switch() method.
  if (!conjoiner->switch(TEMP_MONITOR_TAG, TEMP_MONITOR_IFACE)) {

     // This is a main loop for non-primary replicas where the ft_detector
     // watches a time-out. This is terminated only if a fault is found.
     orb->run();

     // The ft_detector detects a fault. It is now terminated and requests
     // a channel switching to the conjoiner via invoking ::switch() method.
     continue;
  }

  // Primary replica starts here. Publish periodically.
  while(1) {

      // Invoke a method of subscribers.
      monitor->update_temperature('A', value);

      // Publish data to let replicas know my aliveness.
      ft_monitor->I_am_alive();

      if (conjoiner->is_switched()) break;
  }
}
```

Fig. 10. Passively replicated publisher code.

in [12]. Such a replication policy significantly helped reduce the complexity of our fault tolerant CAN-CORBA design. Since the fault tolerant CAN-CORBA provides replications for publishers, subscribers, and a conjoiner, not only can programmers be free from the single point of failure caused by the centralized conjoiner, but also they can freely add fault tolerance to their designs through replicating CAN-CORBA objects. Finally, we showed a program example that made use of the proposed fault tolerance features.

Although we are still implementing and evaluating the fault tolerant CAN-CORBA, we strongly believe that additional resource requirements of the fault tolerant CAN-CORBA fall in a reasonable boundary where most CAN based embedded systems could handle. The new CAN-CORBA design demonstrated that it was feasible to use a fault tolerant CORBA in developing distributed embedded systems on real-time networks with severe resource limitations.

7 Acknowledgements

Thanks to the referees and others who provided us with feedback about the paper. Special thanks to Seoul National University RTOS Lab. members who have participated in a series of heated discussions through a year long seminar on this topic.

The work reported in this paper was supported in part by MOST under the National Research Laboratory grant and by Automatic Control Research Center. T.-H. Kim was supported in part by '99 RIET (Research Institute for Engineering and Technology) grant and research fund for new faculty members from Hanyang University, Korea.

References

1. Bosch: CAN Specification, Version 2.0. (1991)
2. Cristian, F.: Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. Distributed Computing, Vol. 4. 1991) 175–187
3. Fetzer, C.: Fail-Aware Publish/Subscribe Communication in Erlang. Proceedings of the 4th International Erlang User Conference. (1998)
4. Object Management Group: Fault-Torerant CORBA Using Entity Redundancy Request For Proposal, OMG Document orbos/98-04-01 edition. (1999)
5. Object Management Group: Minimum CORBA - Joint Revised Submission , OMG Document orbos/98-08-04 edition. (1998)
6. Object Management Group: Fault-Torerant CORBA - Joint Revised Submission, OMG TC Document orbos/99-12-08 edition. (1999)
7. ISO-IS 11898: Road vehicles - Interchange of digital information - Controller Area Network (CAN) for high speed communication. (1993)
8. Kaiser, J., Mock, M.: Implementing the Real-Time Publisher/Subscriber Model on the Controller Area Network (CAN). IEEE International Symposium on Object-oriented Real-time distributed Computing. (1999)
9. Kim, K., Jeon, G., Hong, S., Kim, S., Kim, T.: Resource Conscious Customization of CORBA for CAN-based Distributed Embedded Systems. IEEE International Symposium on Object-Oriented Real-Time Computing. (2000)

10. Kim, K., Jeon, G., Hong, S., Kim, T., Kim, S.: Integrating Subscription-based and Connection-oriented Communications into the Embedded CORBA for the CAN Bus. IEEE Real-time Technology and Application Symposium. (2000)
11. Oki, B., Pfluegl, M., Siegel, A., Skeen, D.: The Information Bus – An Architecture for Extensible Distributed Systems. ACM Symposium on Operating System Principles. (1993)
12. Rajkumar, R., Gagliardi, M.: High Availability in the Real-Time Publisher/Subscriber Inter-Process Communication Model. IEEE Real-Time Systems Symposium. (1996)
13. Rajkumar, R., Gagliardi, M., Sha, L.: The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation. IEEE Real-time Technology and Application Symposium. (1995)

A Real-Time Animator for Hybrid Systems

Tobias Amnell[1], Alexandre David[1], and Wang Yi[1]

Department of Computer Systems
Uppsala University
{tobiasa,adavid,yi}@docs.uu.se

Abstract. In this paper, we present a real time animator for dynamical systems that can be modeled as hybrid automata i.e. standard finite automata extended with differential equations. We describe its semantic foundation and its implementation in Java and C using CVODE, a software package for solving ordinary differential equations. We show how the animator is interfaced with the UPPAAL tool to demonstrate the real time behavior of dynamical systems under the control of discrete components described as timed automata.

1 Introduction

UPPAAL is a software tool for modeling, simulation and verification of real time systems that can be described as timed automata. In recent years, it has been applied in a number of case studies [6,7,8,4,3], which demonstrates the potential application areas of the tool. It suits best the class of systems that contain only discrete components with real time clocks. However it can not handle more general hybrid systems, which has been a serious restriction on many industrial applications. The goal of this work is to extend the UPPAAL tool with features for modeling and simulation of hybrid systems.

A hybrid system is a dynamical system that may contain both discrete and continuous components whose behavior follows physical laws [5], e.g. process control and automotive systems. In this paper, we shall adopt hybrid automata as a basic model for such systems. A hybrid automaton is a finite automaton extended with differential equations assigned to control nodes, describing the physical laws. Timed automata [1] can be seen as special class of hybrid automata with the equation $\dot{x} = 1$ for all clocks x. We shall present an operational semantics for hybrid automata with dense time and its discrete version for a given time granularity. The discrete semantics of a hybrid system shall be considered as an approximation of the continuous behavior of the system, corresponding to sampling in control theory.

We have developed a real time animator for hybrid systems based on the discrete semantics. It can be used to simulate the dynamical behavior of a hybrid system in a real time manner. The animator implements the discrete semantics for a given automaton and sampling period, using the differential equation solver CVODE. Currently the engine of the animator has been implemented in Java and C using CVODE. We are aiming at a graphical user interface for editing and

J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 134–145, 2001.

showing moving graphical objects and plotting curves. The graphical objects act on the screen according to physical laws described as differential equations and synchronize with controllers described as timed automata in UPPAAL.

The rest of the paper is organized as follows: In the next section we describe the notion of hybrid automata, the syntax and operational semantics. Section 3 is devoted to implementation details of the animator. Section 4 concludes the paper.

2 Hybrid Systems

A hybrid automaton is a finite automaton extended with differential equations describing the dynamical behavior of the physical components.

2.1 Syntax

Let X be a set of real-valued variables X ranged over by x, y, z etc including a time variable t.

We use $\dot{x}$ to denote the derivative (rate) of x with respect to the time variable t. This variable represents the global time. Note that in general $\dot{x}$ may be a function over X; but $\dot{t} = 1$. We use $\dot{X}$ to stand for the set of differential equations in the form $\dot{x} = f(X)$ where f is a function over X.

Assume a set of predicates over the values of X; for example, $2^x + 1 \leq 10$ is such a predicate. We use $\mathcal{G}$ ranged over by g, h etc to denote the set of boolean combinations of the predicates, called *guards*.

To manipulate variables, we use concurrent assignments in the form: $x_1 := f_1(X) \ldots x_n := f_n(X)$ that takes the current values of the variables X as parameters for f_i and updates all x_i's with $f_i(X)$'s simultaneously. We use Γ to stand for the set of concurrent assignments.

We shall study networks of hybrid automata in which component automata synchronize with each other via complementary actions. Let $\mathcal{A}$ be a set of action names. We use $\mathcal{A}ct = \{\ a? \mid \alpha \in \mathcal{A}\ \} \cup \{\ a! \mid \alpha \in \mathcal{A}\ \} \cup \{\ \tau\ \}$ to denote the set of actions that processes can perform to synchronize with each other, where τ is a distinct symbol representing internal actions.

A hybrid automaton over X, $\dot{X}$, $\mathcal{G}$, $\mathcal{A}ct$ and Γ is a tuple $\langle L, E, I, T, l_0, X_0 \rangle$ where

- L is a finite set of names standing for control nodes.
- E is the equation assignment function: $E : L \to 2^{\dot{X}}$.
- I is the invariant assignment function: $I : L \to \mathcal{G}$ which for each node l, assigns an invariant condition $I(l)$.
- T is the transition relation: $T \subseteq L \times (\mathcal{G} \times \mathcal{A}ct \times \Gamma) \times L$. We denote $(l, g, \alpha, \gamma, l')$ by $l \to g, \alpha, \gamma l'$. For simplicity, we shall use $l \to g, \gamma l'$ to stand for $l \to g, \tau, \gamma l'$.
- $l_0 \in L$ is the initial node.
- X_0 is the initial variable assignment.

To study networks of automata, we introduce a CCS-like parallel composition operator. Assume that $A_1, ..., A_n$ are automata. We use $\overline{A}$ to denote their parallel composition. The intuitive meaning of $\overline{A}$ is similar to the CCS parallel composition of $A_1, ..., A_n$ with *all* actions being restricted, that is, $\overline{A} = (A_1|...|A_n)\backslash \mathcal{A}ct$. Thus only synchronization between the components A_i is possible. We call $\overline{A}$ a *network of automata*. We simply view $\overline{A}$ as a vector and use A_i to denote its ith component.

Example 1. In figure 1 we give a simple example hybrid automaton that describes a bouncing ball and a touch sensitive floor. The left automaton defines three variables, x, the horizontal distance from the starting point, the height y and the speed upwards u. These variables can be observed by the other automata. Initially the x-speed is 1, the ball is at 20 m height and the gravitational constant is 9.8. The variables will change according to their equations until the transition becomes enabled when $y <= 0$. The middle automaton is a model of a sensor that will issue a signal when the ball hits the floor.

The rightmost automaton is a controller of the system. It does not have any differential equations, only clocks, and is a proper timed automata as used in the Uppaal tool. It synchronizes with the sensor signal (bounce!) and resets a clock z. If the intervals between signals are longer than 5 time units (i.e. z¿5) it will return to the initial location. But if the clock is less than 5 (that is the intervals between bounces is shorter than 5) the automaton will go to the location *low_bounces*.

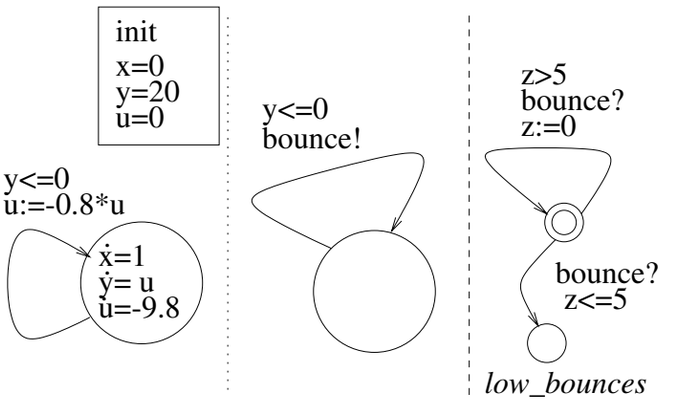


Fig. 1. Bouncing ball with touch sensitive floor and control program.

Example 2. As a more complex example we show a model of an industrial robot. In figure 2 a schematic view of a robot with a jointed arm is shown. The inner

arm can be turned 360 degrees around the z-axis, it can also be raised and lowered between 40 and 60 degrees. The outer arm is positioned at the tip of the inner arm and can be raised and lowered.

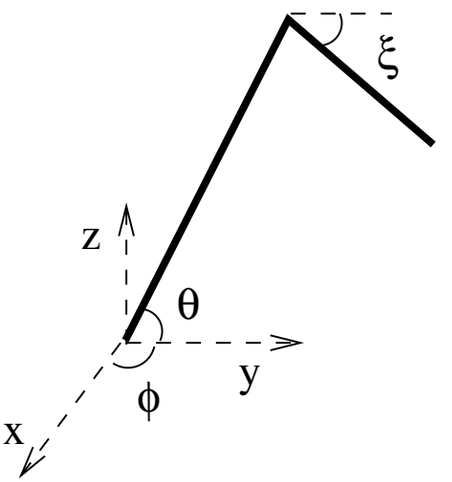


Fig. 2. Industrial robot with three degrees of freedom.

In figures 3 and 4 the hybrid automaton controlling the motion of the robot is shown. We do not show the simple control automaton that starts and stops the execution.

When the execution starts the inner arm will stand still until it receives the `move2pickup?` signal from the control automaton. Then the arm will start turning and lowering the arm so that the gripping tool on the outer arm can reach a table where it will pick up an object. When the inner arm reaches a position in front of the table it will try to synchronize with the outer arms automaton on the signal `pickup!`. After the pickup the robot will raise and turn to another table where it will again try to synchronize with the outer arm on `release!`. After the release the robot will return to the original position and issue the `back!` signal to the controller.

The automaton, in figure 4, controlling the outer arm is simpler since the outer arm only has one degree of freedom. It starts with lowering the arm when the controller sends a `move2pickup2!`. When it comes to the correct angle it will stop and wait for the inner arm to do the same, then they will synchronize on `pickup`. After the pickup the arm will raise until it reaches its max (30 degrees), then it will wait for the inner arm to reach the release position. When the robot is at the release table the outer arm will descend and then release the object in its grip. On the return to the original position the outer arm will rise again.

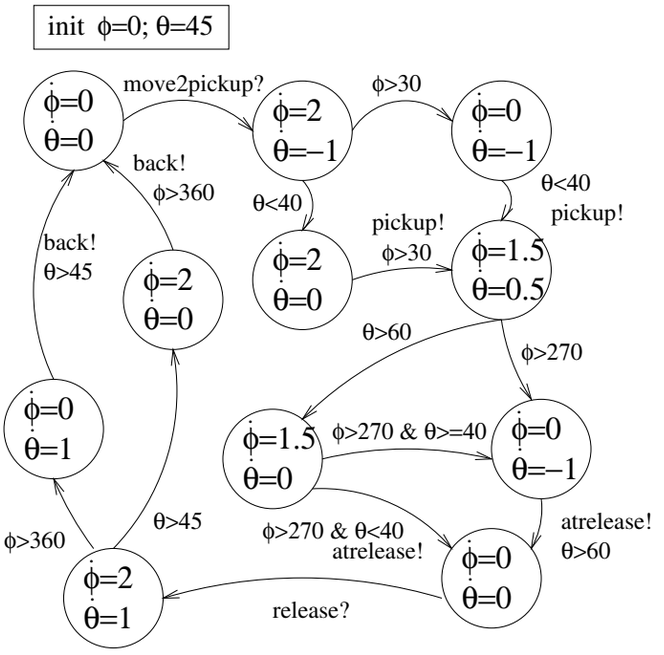


Fig. 3. Robot inner arm automaton

2.2 Semantics

To develop a formal semantics for hybrid automata we shall use variable assignments. A *variable assignment* is a mapping which maps variables X to the reals. For a variable assignment σ and a delay Δ (a positive real), $\sigma + \Delta$ denotes the variable assignment such that

$$(\sigma + \Delta)(x) = \sigma(x) + \int_{\Delta} \dot{x} dt$$

For a concurrent assignment γ, we use $\gamma[\sigma]$ to denote the variable assignment σ' with $\sigma'(x) = Val(e, \sigma)$ whenever $(x := e) \in \gamma$ and $\sigma'(x') = \sigma(x')$ otherwise, where $Val(e, \sigma)$ denotes the value of e in σ. Given a guard $g \in \mathcal{G}$ and a variable assignment σ, $g(\sigma)$ is a boolean value describing whether g is satisfied by σ or not.

A *node vector* $\bar{l}$ of a network $\overline{A}$ is a vector of nodes where l_i is a location of A_i. We write $\bar{l}[l'_i/l_i]$ to denote the vector where the ith element l_i of $\bar{l}$ is replaced by l'_i.

A *state* of a network $\overline{A}$ is a configuration $(\bar{l}, \sigma)$ where $\bar{l}$ is a node vector of $\overline{A}$ and σ is a variable assignment.

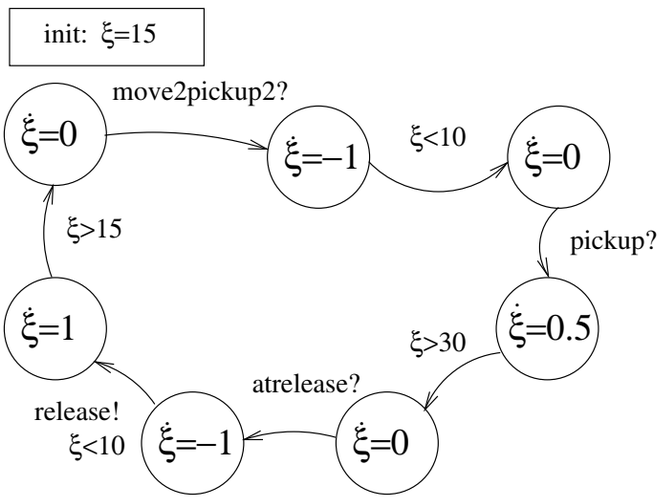


Fig. 4. Robot outer arm automaton

The *semantics of a network* of automata $\overline{A}$ is given in terms of a labelled transition system with the set of states being the configurations. The transition relation is defined by the following three rules:

- $(\bar{l}, \sigma) \overset{\alpha}{\rightsquigarrow} (\bar{l}[l'_i/l_i], \gamma_i[\sigma])$ if $l_i \overset{g_i\ \alpha,\gamma_i}{\longrightarrow} l'_i$ and $g_i(\sigma)$ for some l_i, g_i,α,γ_i.
- $(\bar{l}, \sigma) \overset{\tau}{\rightsquigarrow} (\bar{l}[l'_i/l_i, l'_j/l_j], (\gamma_j \cup \gamma_i)[\sigma])$ if $l_i \overset{g_i\ a!\gamma_i}{\longrightarrow} l'_i$, $l_j \overset{g_j\ a?\gamma_j}{\longrightarrow} l'_j$, $g_i(\sigma)$, $g_j(\sigma)$, and $i \neq j$, for some l_i, l_j, g_i, g_j, a, γ_i, γ_j.
- $(\bar{l}, \sigma) \overset{\Delta}{\rightsquigarrow} (\bar{l}, \sigma + \Delta)$ if $I(\bar{l})(\sigma)$ and $I(\bar{l})(\sigma + \Delta)$ for all positive real numbers Δ.

where $I(\bar{l}) = \bigwedge_i I(l_i)$.

The execution of a hybrid automata then becomes an alternating sequence of delay and action transitions in the form:

$s_0 \overset{\Delta_0}{\rightsquigarrow} (\bar{l}_0, \sigma_0 + \Delta_0) \overset{\alpha_0}{\rightsquigarrow} (\bar{l}_1, \sigma_1) \overset{\Delta_1}{\rightsquigarrow} (\bar{l}_1, \sigma_1 + \Delta_1) \overset{\alpha_1}{\rightsquigarrow} (\bar{l}_2, \sigma_2) \ldots$
$(\bar{l}_i, \sigma_i) \overset{\Delta_i}{\rightsquigarrow} (\bar{l}_i, \sigma_i + \Delta_i) \overset{\alpha_i}{\rightsquigarrow} (\bar{l}_{i+1}, \sigma_{i+1})$

2.3 Tick Semantics

The operational semantics above defines how an automaton will behave at every real-valued time point with arbitrarily fine precision. In fact, it describes all the possible runnings of a hybrid automaton.

In practice, a "sampling" technique is often needed to analyze a system. Instead of examining the system at *every* time point, which often is impossible, only a finite number of time points are chosen to approximate the full system behavior. Based on this idea, we shall adopt a time–step semantics called δ–semantics relativized by the granularity δ, which describes how a hybrid system

shall behave in every δ time units. In practical applications, the time granularity δ is chosen according to the nature of the differential equations involved. In a manner similar to sampling of measured signals the sampling interval should be short for rapidly changing functions. To achieve finer precision, we can choose a smaller granularity.

We use the distinct symbol χ to denote the sampled time steps. Now we have a discrete semantics for hybrid automata.

- $(\bar{l}, \sigma) \overset{\chi}{\mapsto} (\bar{l}, \sigma + \delta)$ if $(\bar{l}, \sigma) \overset{\delta}{\rightsquigarrow} (\bar{l}, \sigma + \delta)$ and
- $(\bar{l}, \sigma) \overset{\alpha}{\mapsto} (\bar{l}', \sigma')$ if $(\bar{l}, \sigma) \overset{\alpha}{\rightsquigarrow} (\bar{l}', \sigma')$

We use β_i to range over $\{\chi, \tau\}$ representing the discrete transitions. The "sampled" runs of a hybrid automaton will be in the form:

$(\bar{l}_0, \sigma_0) \overset{\beta_1}{\mapsto} (\bar{l}_1, \sigma_1) \ldots (\bar{l}_i, \sigma_i) \overset{\beta_{i+1}}{\mapsto} (\bar{l}_{i+1}, \sigma_{i+1}) \ldots$

In the following section, we shall present a real time animator based on the δ-semantics. For a given hybrid automaton, the animator works as an interpretor computing the δ-transitions step by step using CVODE, a differential equation solver.

3 Implementation

Our goal is to extend the Uppaal tool to deal with hybrid systems. We divide the system into two parts: one "hybrid" part where we use full hybrid automata to describe the components; and one "control" part where we restrict the automata to timed automata as accepted by Uppaal. The two parts do not, in the current implementation, share any variable and communicate solely through synchronization channels. Concerning the time, the time measured by the clocks on the Uppaal side is the same as the time on the hybrid side.

Each animator object, i.e. a hybrid automaton, is associated with one Uppaal automaton. This automaton is used as a link to Uppaal. It is an abstraction of the environment model in the sense that it can produce the same events as the environment. More formally it simulates the behavior of the hybrid automaton, the visible actions being the events. This choice of implementation is motivated by the desire to model-check the rest of the Uppaal model, i.e. the controller part, as a closed system. Figure 5 shows the association of animator objects with Uppaal automata.

The approach we use is to create a new module that interacts with the present Uppaal toolkit, and to use an existing and proven solver for the differential equations. The Uppaal GUI is written in Java and the differential equation solver that we have adopted, CVODE, is written in C. This gives the natural architecture of the animator: the animator itself with the objects is written in Java and the engine of the animator in C, connected through the Java native interface (JNI). The two main layers of the implementation are the animation system and the CVODE layers.

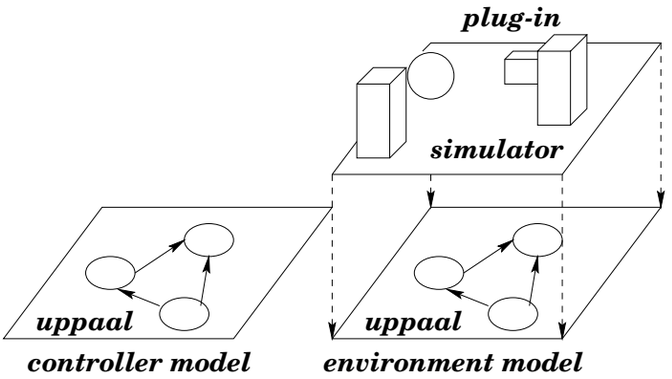


Fig. 5. Association between animator objects and UPPAAL automata.

3.1 The Animation System Layer

The system to be modeled is defined as a collection of *objects*. Each object is described by a hybrid automaton with its corresponding variables. Every state of the hybrid automaton has a set of equations and transitions. The equations, conditions (guards) and assignments are given as logical/arithmetic expressions with ordinary mathematical functions such as sine, cosine ..., and also user defined functions. In figure 6 the input in textual format corresponding to example 1 is shown.

The first part of the animation object description consists of variable initialisations. All variables updated by this automaton must be initialized. The initialization must be a constant but it can be expressed as a function that is evaluated to a constant, (e.g. `a=sin( pi/3 );`). Then there may optionally be definitions of user functions, such as `sin2(x)=sin(x)^2;` or `diff(x,y)=abs(x-y);`. The nodes of the automaton are all named and set out as `:name:`. For each node the differential equations are given, e.g. `x' = sin2(y);`, and the transitions in form of if-expressions. The syntax for the transitions are: `if (` *condition* `) {` *action* `}` *node* `;`.

3.2 The CVODE Layer

At the heart of the animator we have used the CVODE [2] solver for ordinary differential equations (ODE's). This is a freely available ODE solver written in C, but based on two older solvers in Fortran.

The mathematical formulation of an initial value ODE problem is

$$\dot{x} = f(t, x),\ x(t_0) = x_0,\ x \in \mathbb{R}R^N. \tag{1}$$

Note that the derivative is only first order. Problems containing higher order differential equations can be transformed to a system of first order. When using

```
// Bouncing ball object
u = 0;
x = 0;
y = 20;

:bounce1:
u' = -9.8;
x' = 1;
y' = u;

if (y<=0) { u=-0.8*u } bounce1;
```

```
// Touch sensor
:wait:
if (y<=0) {bounce!;} wait;
```

Fig. 6. Textual input corresponding to bouncing ball in figure 1

CVODE one gets a numerical solution to (1) as discrete values x_n at time points t_n.

CVODE provides several methods for solving ODE's, suitable for different types of problem. But since we aim at general usage of the animator engine we cannot assume any certain properties of the system to solve. Therefore we only use the full dense solver and assume that the system is well behaved (non-stiff in numerical analysis terminology). This will give neither the most memory efficient nor the best solution, but the most general.

We use one CVODE solver for the whole system. This is set up and started with new initial values at the beginning of each delay transition. The calculations are performed stepwise, one "tick" (δ–transition) at a time. After each tick all the conditions of the current state are checked, if any is evaluated to true one has to be taken. If an assignment on the transition changes a variable the solver must be reinitialized before the calculations can continue.

It is worth pointing out that the tick length δ is independent of the internal step size used by the ODE solver, the solver will automatically choose an appropriate step size according to the function calculated and acceptable local error of the computation. From the solvers point of view the tick intervals can be seen as observation or sampling points.

After each tick the system variables are returned to the Java side of the animator where they are used either to update a graph or as an input to move graphical objects.

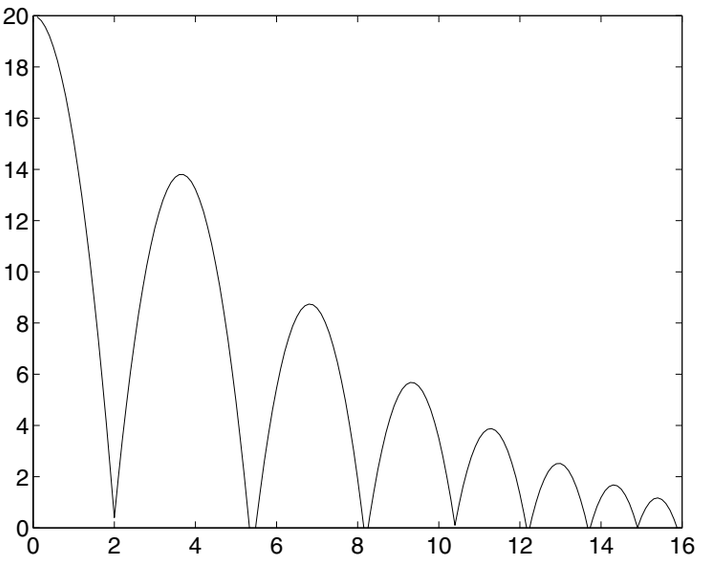


Fig. 7. Bouncing ball on touch sensitive floor that continues until bounces are shorter than 1 second

Example 1, continued. In figure 7 a plot of the system described in figure 1 is shown. The plot shows the height and the distance of the bouncing ball. Not shown in the figure is that the touch sensitive floor will create a signal every time the ball hits the floor, and that the system will continue running until the time between bounces is less than 1 second.

Example 2, continued. For the robot example we only show, in figure 8, how the outer arm will raise and turn during the execution of the system. In order to get a better view, the plot only shows the movement from the initial position to the release. The robot starts at the right and turns counter clock-wise.

4 Conclusion

We have presented a real time animator for hybrid automata. For a given hybrid automaton modeling a dynamical system and a given time granularity representing sampling frequency, the animator demonstrates a possible running of the system in real time, which is a sequence of sampled transitions. The animator has been implemented in Java and C using CVODE, a software package for solving differential equations. As future work, we aim at a graphical user interface for editing and showing moving graphical objects and plotting curves. The graphical objects act on the screen according to the differential equations and synchronize with controllers described as timed automata in Uppaal.

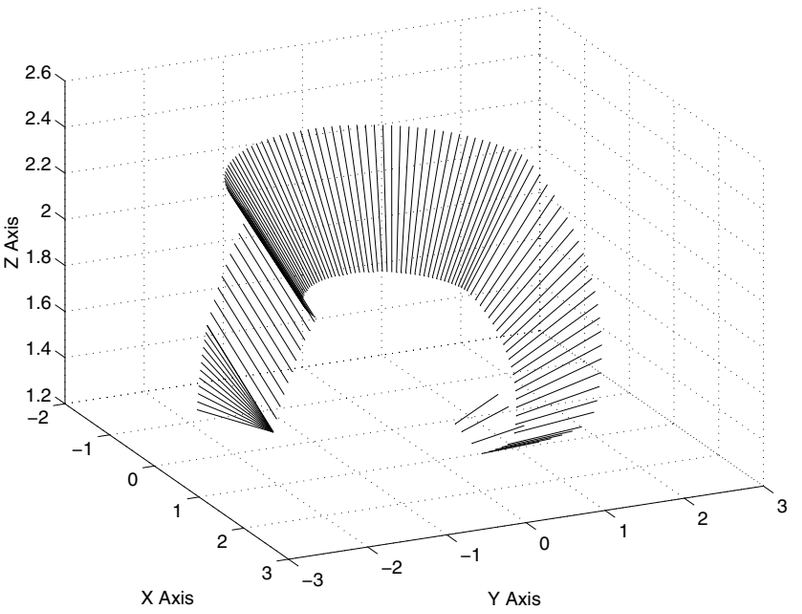


Fig. 8. The movement of the robots outer arm.

References

[1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 125:183–235, 1994.

[2] S. Cohen and A. Hindmarsh. Cvode, a stiff/nonstiff ode solver in c. *Computers in Physics*, 2(10):138–43, March-April 1996.

[3] P. D'Argenio, J.-P. Katoen, T. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In *Proceedings of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *LNCS*, pages 416–431. Springer-Verlag, April 1997. Enschede, The Netherlands.

[4] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal modelling and analysis of an audio/video protocol: An industrial case study using uppaal. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 2–13. IEEE, December 1997. San Francisco, California, USA.

[5] T. A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic on Computer Science (LICS 96)*, pages 278–292. IEEE, 1996.

[6] K. J. Kristoffersen, K. G. Larsen, P. Pettersson, and C. Weise. Experimental batch plant - vhs case study 1 using timed automata and uppaal. Deliverable of EPRIT-LTR Project 26270 VHS (Verification of Hybird Systems), 1999.

[7] M. Lindahl, P. Pettersson, and W. Yi. Formal design and analysis of a gear controller. In B. Steffen, editor, *Proceedings of the 4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *LNCS*, pages 281–297, 1999. Gulbenkian Foundation, Lisbon, Portugal.

[8] H. Lönn and P. Pettersson. Formal verification of a tdma protocol start-up mechanism. In *Proceedings of 1997 IEEE Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 235–242. IEEE, December 1997. Taipei, Taiwan.

Reordering Memory Bus Transactions for Reduced Power Consumption

Bruce R. Childers and Tarun Nakra

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15208
{childers, nakra}@cs.pitt.edu

Abstract. Low energy consumption is becoming the primary design consideration for battery-operated and portable embedded systems, such as personal digital assistants, digital still and movie cameras, digital music playback, medical devices, etc. For a typical processor-based system, the energy consumption of the processor-memory component is split roughly 40-60% between the processor and memory. In this paper, we study the impact of reordering memory bus traffic on reducing bus switching activity and power consumption. To conduct this study, we developed a software tool, called MPOWER, that lets an embedded system designer collect a trace of memory bus accesses and determine the switching activity of the trace, given design parameters such as data and address bus width, bus multiplexing, cache size, block size, etc. Using MPOWER, we measured the effectiveness of reordering memory accesses on switching activity. We found that for small caches, which are typical of embedded processors, the number of signal transitions in an ideal case can be reduced by an average of 53%. This paper also describes a practical hardware scheme for reordering the elements within a cache line to reduce switching activity. We found that cache line reordering reduces switching activity by 15–31%.

1 Introduction

In the past decade, the demand for small, battery operated embedded systems, such as cellular phones, digital motion and still cameras, personal digital assistants (PDAs), medical devices, etc., has seen explosive growth. For example, in 1997, 163 million cellular phones were shipped worldwide with market growth expected to reach 356 million units in 2000 [21]. Not only has the sheer demand for battery-operated devices grown, but the expectations of those devices has also increased dramatically. As the capabilities of handheld portable systems continue to increase, there is an ever greater need for more computing power with extended battery life. Indeed, in the cellular phone and laptop computer markets, processing capabilities and battery life are two of the most important features that manufacturers use to differentiate their products.

Although aggressive processors such as Hewlett-Packard's PA-8500 [12] and Compaq's Alpha 21264 [13] have the processing capabilities demanded by high-performance embedded applications, they are not suitable for small portable devices due to high power consumption and heat dissipation (for example, the Alpha 21264

J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 146-161, 2001.

dissipates 72 watts of power [11]). To solve this problem, new processors are being developed which have high performance, while offering low power consumption [9,22].

This paper studies the impact of reordering memory bus transactions on the switching activity and power consumption of the memory system. We developed a tool set, called MPOWER, for evaluating the impact of memory bus design on power consumption. MPOWER lets an embedded system designer vary the underlying bus protocol and structure to determine its effect on the power/energy budget. In this paper, we use MPOWER to answer the question of whether reordering memory bus transactions is worthwhile for embedded processors, and in what situations to apply it. We also briefly outline in this paper a hardware scheme for reordering a cache line to reduce switching activity based on the encouraging results from our initial experiments.

This paper is organized as follows. Section 2 presents background material on power consumption, and Section 3 describes the goal of bus transaction reordering. Section 4 presents a detailed performance analysis of the effect of different memory bus structures and cache sizes on the ability of bus reordering to reduce switching activity. Section 5 briefly sketches an application of reordering to the data elements of a cache line and how reordering within a cache line can reduce switching activity. Section 6 describes related work, and Section 7 concludes the paper.

2 Power Consumption

The amount of power consumed by a portable system is a useful metric for determining the maximum current that a battery must supply. It is also useful for determining the packaging features needed to handle an embedded processor's expected heat dissipation. Along with power consumption, a device's energy should be considered because it measures power over time and determines battery life.

We need to be careful that architectural features that reduce power consumption do not increase execution latency so much that there is no net decrease in energy consumption. In CMOS circuits, the power consumption due to dynamic switching (so called "bit flips") dominates the power lost to static leakage [5]. The average power of a CMOS device, P_{avg}, can be expressed in terms of its switching activity:

$$P_{avg} = \alpha_T \bullet f_{CLK} \bullet C_{load} \bullet V^2_{DD}$$

where α_T is the switching activity factor (i.e., the average number of signal transitions per clock cycle), f_{CLK} is the clock frequency, C_{load} is the capacitance load, and V_{DD} is the supply voltage. Assuming that f_{CLK}, C_{load}, and V_{DD} are held constant, switching activity determines the power consumption of a device.

In the case of off-chip resources, the capacitance load is very large relative to on-chip resources, and high C_{load} leads to high power consumption (i.e., the drivers for external pins are very large and consume much power when driving long wires with high capacitance). Thus, reducing the average number of signal transitions when driving external pins reduces overall energy consumption, assuming execution latency is not increased. For the memory bus, which accounts for 15-30% of the memory hierarchy's energy consumption [10], reordering bus transactions to minimize

switching activity may significantly reduce the energy consumption of the memory hierarchy. This paper focuses on reducing α_T for the external memory bus. Improving α_T corresponds to reducing the number of signal transitions that occur over a program's execution lifetime, and for this paper, we use the reduction in switching activity as our metric to indicate reduced power consumption.

3 Memory Bus Reordering

Because off-chip memory accesses are very expensive power-wise, we are studying how much reordering bus transactions (to minimize signal transitions) reduces overall energy consumption. One strategy for reducing the number of bit flips on the memory bus is to schedule bus transactions (a bus transaction is a read or write of a single word) in the order in which they would cause the minimal signal changes. Of course, the dependences between bus transactions must be taken into account. For example, a load that follows a store in program execution order can not be scheduled before the store (assuming they use the same address). Likewise, an instruction that causes a data cache line replacement must be scheduled before the transactions that replace the line.

To investigate the effect of reordering bus transactions on energy consumption, we

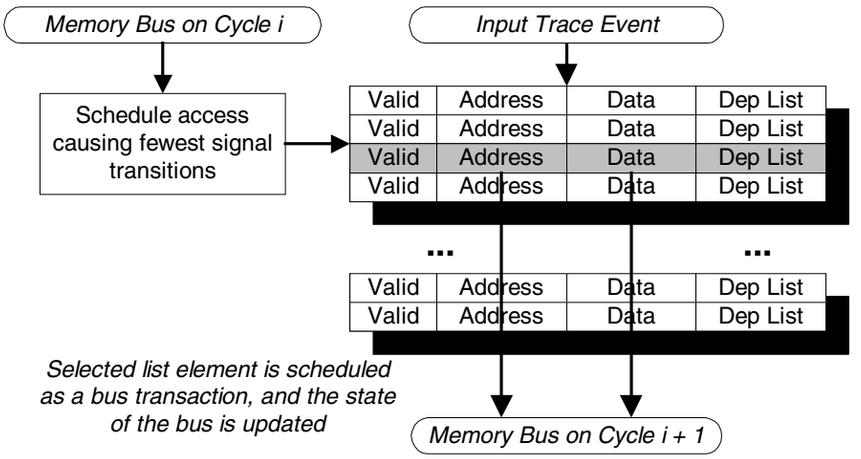


Fig. 1. Transaction scheduler

incorporated a *transaction scheduler* in MPOWER. The scheduler maintains a list of recently seen off-chip memory access events (e.g., a request to load a cache line), and schedules bus transactions from the list according to which transaction causes the least bit flips given the current state of the bus. The scheduler ensures that all dependences between instructions and memory reads and writes are satisfied by building the dependence graph on-the-fly and only scheduling transactions that are leaf nodes. The transaction scheduler uses *priority list scheduling*, where the priority is the number of bit transitions between successive bus transactions.

Figure 1 illustrates the operation of the scheduler. The scheduler accepts an input event from a memory access trace, and places it in the scheduling list. If the new event causes the list to reach a threshold size, an event is removed from the list and scheduled as a bus transaction. Only those events that are have no dependences with other events in the list may be scheduled as an output transaction. Of the ready events, the one that causes the least number of signal transitions is chosen.

The scheduler is locally greedy and may not find an optimal order for the whole trace. However, with large thresholds, the scheduler gives a good approximation that indicates the worth of reordering transactions. Furthermore, we do not use the scheduler as a hardware device to reorder memory bus transactions. Instead, we use the scheduler to answer the question of whether reordering is valuable, and if so, in what situations to apply it. We are currently studying hardware mechanisms and compiler support for transaction reordering. Section 5 presents preliminary results in this regard.

4 Experimental Results

To study the effect of memory bus reordering on switching activity, we used several applications from the MediaBench [16] and SPEC'95 benchmark suites. These applications are described in Table 1 (the ones marked with † are from SPEC'95). The benchmarks were chosen to be representative of common applications for communication devices and PDAs.

MPOWER uses a modified version of the SimpleScalar processor simulator from the University of Wisconsin [4]. SimpleScalar was instrumented to collect a trace of memory accesses between the L1 instruction and data caches and off-chip memory.

Table 1. Benchmark applications

Benchmark	Description
adpcm	Audio decoder
gsm	Full-rate speech decoder
g721	Voice compression
jpeg	Image compression
perl†	Perl interpreter
li†	Lisp interpreter
go†	Game of GO

Figure 2 shows the structure of a typical processor-memory interface [8]. Our simulated architecture has small on-chip separate instruction and data caches, with a narrow bus between the caches and the off-chip DRAM memory controller. In the experiments that follow, we evaluate different bus structures (multiplexing and bus width) and their impact on reordering memory transactions. We first examine the general issues associated with reordering, and based on our experiments, we briefly

describe in the second portion of the paper a simple hardware scheme for arranging the elements of a cache line to minimize signal transitions.

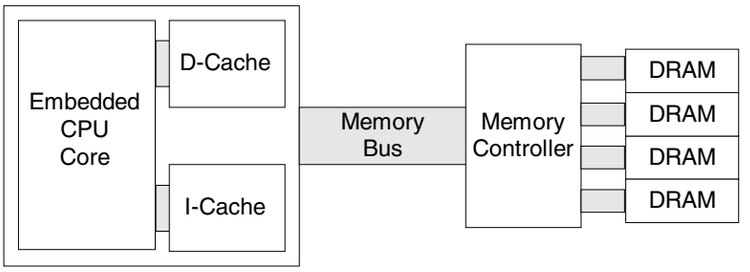


Fig. 2. Processor-memory system structure

In the initial experiments, we assume that the external memory bus has no bidirectional signals; i.e., there are separate data input and output lines. For unidirectional buses, a signal is always driven and will only experience a change in state when a different bit value is written. We use a unidirectional bus for our initial experiments because it gives the most scheduling freedom and lets us best answer the question of how well reordering transactions reduces switching activity. The experiments in Section 5 use a bus that has bidirectional data lines with tristate drivers, as commonly found in embedded processors such as Motorola's PowerPC 7400 [20]. In this case, the bus is not driven following a burst transfer (i.e., a cache line), and the scheduling scope is limited to a cache line.

Although there is likely to be a latency penalty for reordering bus transactions, in this paper, we make the assumption that the order of transactions does not affect latency. We are currently implementing a form of transaction reordering and investigating its impact on memory latency. In the initial experiments, we assume that the memory system does not do burst transfers, and an individual word in a cache line is transferred in a single transaction. This assumption is relaxed in Section 5 where we consider a bus that supports burst transfers. The experiments in the first part of this paper use a trace of the first 1.5 million off-chip memory accesses. In Section 5, the programs are run to completion.

4.1 Data Width and Multiplexing

The processor-memory bus for an embedded system often has a narrow external interface to reduce cost. For instance, the external memory interface of the Motorola M•CORE MMC2001 embedded processor has a non-multiplexed bus with a data width of 16 bits (two transfers are done to read/write a native word of 32 bits) and an address width of 19 bits [19]. In this section, we examine the effect of bus width and multiplexing on reordering bus transactions and its associated switching activity.

Using MPOWER, we evaluated the impact of the different bus structures that are shown in Figure 3. In the figure, the bus structures have unidirectional connections to

external memory (i.e., data input and output). The low performance bus (LPB) has a bus width of 16 bits that is multiplexed for 32-bit addresses and data. The medium performance bus (MPB) has a bus width of 32 bits that is multiplexed for addresses and data. Finally, the high performance bus (HPB) has separate address and data buses that are 32 bits wide, giving a total bus width of 96 bits.

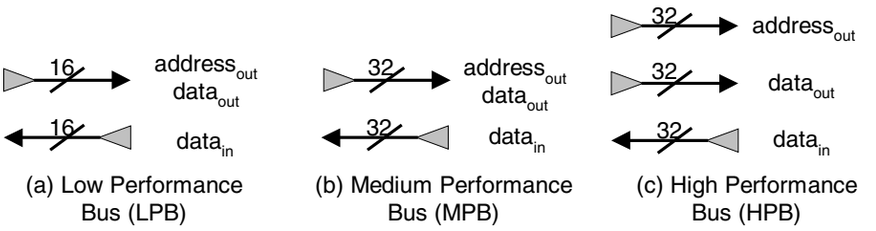


Fig. 3. Bus structures

Figure 4 shows the assumptions made about bus timing. In the case of the LPB, the addresses are placed on the bus starting with the high order address ($a_{31:16}$), followed by the low order address ($a_{15:0}$). Data also appears in the same order for a read or write. The MPB places the full address on the bus prior to reading or writing data. Finally, the HPB places the full address and data on the bus simultaneously for writes. For HPB reads, the data read from memory is available on the next cycle from the data bus.

	LPB		MPB		HPB	
Cycle	Read	Write	Read	Write	Read	Write
0	$a_{31:16}$	$a_{31:16}$	$a_{31:0}$	$a_{31:0}$	$a_{31:0}$	$a_{31:0}$,$d_{31:0}$
1	$a_{15:0}$	$a_{15:0}$	*delay*	$d_{31:0}$	$d_{31:0}$	
2	*delay*	$d_{31:16}$	$d_{31:0}$			
3	$d_{31:16}$	$d_{15:0}$				
4	$d_{15:0}$					

Fig. 4. Bus timing and transaction ordering

Figure 5 shows how bit flip cost is computed for each of the three bus structures. In the cost equations, *pc* is population count (i.e., the number of 1's in a word) and ^ is exclusive- or. The cost computation takes into account the unidirectional data buses and the multiplexing of the address and data bus for writes. For the LPB, the bit flip cost of a write is the number of signal transitions that result between successive portions of the address (high and low halfwords) and the data (high and low halfwords). The cost of a read is computed similarly, except the data cost is computed with respect to the data input bus. Bit costs are computed in a similar way for the other bus structures, taking into account bus width and multiplexing.

		Cost Computation
LPB	Read	$pc(da_{out} \wedge a_{31:16}) + pc(a_{31:16} \wedge a_{15:0}) + pc(d_{in} \wedge d_{31:16}) + pc(d_{31:16} \wedge d_{15:0})$
	Write	$pc(da_{out} \wedge a_{31:16}) + pc(a_{31:16} \wedge a_{15:0}) + pc(a_{15:0} \wedge d_{31:16}) + pc(d_{31:16} \wedge d_{15:0})$
MPB	Read	$pc(da_{out} \wedge a_{31:0}) + pc(d_{in} \wedge d_{31:0})$
	Write	$pc(da_{out} \wedge a_{31:0}) + pc(a_{31:0} \wedge d_{31:0})$
HPB	Read	$pc(a_{out} \wedge a_{31:0}) + pc(d_{in} \wedge d_{31:0})$
	Write	$pc(a_{out} \wedge a_{31:0}) + pc(d_{out} \wedge d_{31:0})$

Fig. 5. Bit flip cost calculation

Figure 6 shows the percentage reduction in switching activity with bus reordering, assuming perfect knowledge about the instruction and data streams (i.e., the scheduler knows the value of words read from memory), and a scheduling window size of 32. The percentages were calculated with respect to a baseline system with the same cache and bus structures, but without transaction reordering. The figure shows three different instruction and data cache sizes from 2K to 8K. Also shown are the LPB, MPB, and HPB structures. In the figure, the reduction in switching activity varies from 9–67% depending on cache size and bus structure.

The figure shows two trends. First, as cache size increases, the relative reduction in switching activity decreases for most of the cases. As expected, with larger caches, the miss rates for the instruction and data accesses is lower, and there are fewer off-chip accesses to main memory. Because the numbers in Figure are relative percentages, they do not always show the decreasing trend in bit flip cost. Although the results are not reported here, the absolute number of bit flips always decreases with an increase in cache size for all benchmarks, even for the HPB (which shows an increasing trend in the figure).

	2K			4K			8K		
Benchmark	**LPB**	**MPB**	**HPB**	**LPB**	**MPB**	**HPB**	**LPB**	**MPB**	**HPB**
adpcm	13.73%	26.36%	36.25%	14.51%	29.64%	43.62%	9.23%	23.41%	48.23%
g721	12.79%	54.62%	55.27%	12.76%	51.96%	52.94%	12.90%	55.00%	55.33%
go	13.73%	40.25%	49.57%	13.01%	39.14%	48.29%	12.33%	40.02%	48.51%
gsm	14.14%	49.08%	52.09%	13.84%	48.20%	50.98%	13.79%	49.68%	52.42%
jpeg	18.60%	45.78%	66.43%	18.92%	43.69%	67.45%	18.56%	42.89%	66.15%
li	16.63%	42.18%	49.17%	15.02%	44.98%	52.76%	13.92%	40.89%	52.22%
perl	17.16%	40.54%	45.62%	16.69%	39.86%	45.45%	14.90%	41.20%	45.89%
Average	15.25%	42.69%	50.63%	14.96%	42.50%	51.64%	13.66%	41.87%	52.68%

Fig. 6. Reduction in switching activity with bus transaction reordering.

The second trend shown in Figure 6 is that the high performance bus benefits the most from bus reordering. The HPB has the largest relative decrease in switching activity because it gives the scheduler the most leeway in choosing transactions. For the LPB, the cost of a transaction is computed over the individual halfwords in the full native word. This has the effect that it causes a large number of bit flips between

upper and lower halfwords, which may have very different bit patterns. For example, the instruction format for a branch typically has a small immediate constant encoded in the lower portion of an instruction with an opcode specified in the upper portion. Hence, there may be a large number of bit changes required between reading successive portions of a branch instruction. For the HPB, the upper opcode bits are not as likely to change between successive instructions. Furthermore, the switching activity is dependent on only adjacent words for the HPB, which the scheduler can arrange to minimize bit flips. For the LPB, the scheduler cannot arrange the halfwords of an instruction or data access to minimize signal transitions. This behavior suggests that the scheduling granularity should match bus width. If the scheduler could order the individual halfwords of an LPB access independently, we expect that there would be a greater decrease in switching activity for the LPB.

From Figure 6, we conclude that bus transaction reordering is most effective for embedded processors which have a split address and data bus with small on-chip caches, but it still helps for other bus structures (LPB and MPB). For the remainder of this paper, we use a HPB and 8K instruction and data caches.

4.2 Scheduling Window Size

Figure 7 shows the percentage reduction in signal transitions for different scheduling window sizes, assuming perfect knowledge about the memory access trace. The figure demonstrates the effect that scheduling window size has on reducing the number of signal transitions.

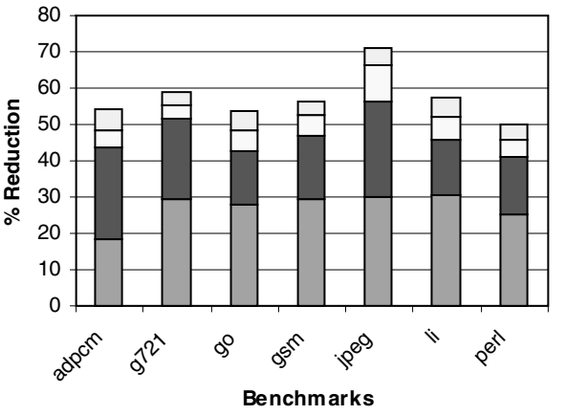


Fig. 7. Percentage reduction in signal transitions for different window sizes.

A window size of 64 does the best overall, ranging from 50–71% reduction, with an average of 57%. However, from the figure, the reduction quickly levels off, and a window size of 16 effectively captures most of the reduction in switching activity. For a window size of 16, the reduction varies from 41–57% with an average of 47%. The

small table size is encouraging since it suggests that a small hardware buffer may capture most of the energy reduction possible through bus transaction reordering.

4.3 Instruction vs. Data Reordering

Because the instruction and data streams have very different properties and behavior, we investigated which stream contributes the most toward reducing bit flips; i.e., for which stream is reordering the most effective, and where should we focus our efforts? Figure 8 shows the percentage of total switching activity that can be attributed to instructions and data. The bottom portion of each bar shows the percentage of total bit flips due to data loads or stores, and the top portion shows the percentage of bit flips due to instruction accesses.

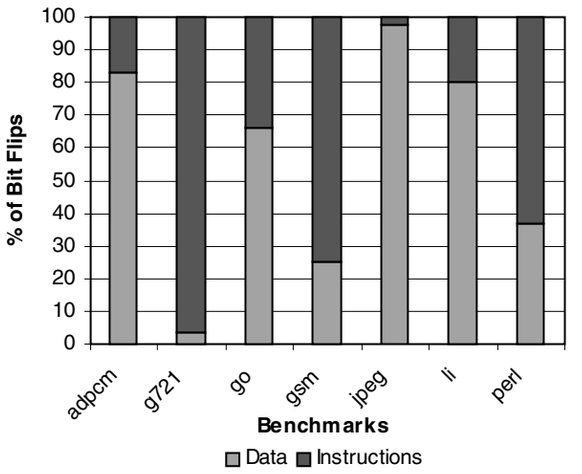


Fig. 8. Percentage of total signal transitions according to type.

In Figure 8, the relative contribution of each stream to switching activity is highly application dependent. For example, *adpcm* and *jpeg* get the most benefit from reordering the data stream because these benchmarks have very low instruction cache miss rates (less than 0.05%). *Adpcm* and *jpeg* have a hot loop that runs directly out of the L1 I-cache, giving a low instruction cache miss rate. Because the on-chip I-cache nicely captures the kernel loop's execution, it causes fewer off-chip accesses, and the data stream dominates switching activity (the L1 D-cache miss rate for *adpcm* and *jpeg* is 2.7% and 1.3%). G*721* and *gsm* have different behavior: they are dominated by instruction accesses, which means that reordering the instruction stream is most effective.

Not only is reordering dependent on cache miss rates, it is also dependent on data and address density. For an application which has dense data (i.e., minimal bit changes between successive data items), the data stream will likely contribute less to the overall reduction in switching activity. Also, the encoding of the instruction set

architecture can have a large impact on switching activity and energy consumption of the memory hierarchy [3,18]. An encoding that ensures the smallest number of bit changes for common code sequences is the most effective at reducing switching activity.

Assuming that we have an efficient hardware scheme to do reordering, then Figure 8 indicates that we should adjust the reordering scheme to match an application's behavior. For example, *adpcm* does not need data reordering, so we could turn off reordering for the data stream to reduce the energy consumption of the reordering hardware. I.e., the energy saved by reordering bus transactions for the data stream may not be worth the energy spent computing the best order to access memory. Similarly, if we are building an application- specific processor, we may leave out instruction or data reordering depending on the application's behavior to reduce chip area. Finally, for a system with a mixed workload, including both data and instruction reordering is important. In this case, as suggested above, we will likely want to turn on/off the reordering hardware on a per application basis using dynamic techniques (e.g., counters for cache miss rates) or profile information.

4.4 Reordering with Real Knowledge

The experiments in the previous sections assume perfect knowledge about the instruction and data streams. Having perfect knowledge about the data stream means that the scheduler knows the value of data reads. Perfect knowledge of the instruction stream means that the scheduler knows the value of the loaded instructions as well as the execution paths within the program. The scheduler takes into consideration the value of a data or instruction read when deciding which transaction to schedule. In practice, the values of loaded items would not be available while reordering transactions. Also, since the scheduler is trace driven, knowledge of execution paths is implicit within the traces. However, the previous experiments give an upper bound on how much we can reduce switching activity of the memory bus. Section 5 presents the effects of a cache line reordering technique on switching activity by actually executing the programs.

In this section, we relax the assumption about perfect knowledge, and consider the case where the scheduler only knows execution paths and the value of stores and instructions. Figure 9 shows the effect of our relaxed assumptions as a decrease in switching activity over a baseline that does not reorder bus transactions. The first set of bars shows the decrease in switching activity when the scheduler only has knowledge about execution paths and stored values, and the second set of bars shows the decrease in switching activity when the scheduler has knowledge about execution paths, stored values, and instruction values. The latter case is shown to demonstrate the potential of instruction reordering when the compiler is able to rearrange the instructions.

In Figure 9, the reduction in switching activity for reordering with knowledge about execution paths and stored values varies from 15–57%, with an average of 29%. *G721* and *gsm* have small decreases in switching because the majority of their accesses are associated with instruction or data reads. Nevertheless, even in this case, it is beneficial to reorder only stores, especially since the hardware mechanisms for reordering stores are likely to be very simple (e.g., a store buffer).

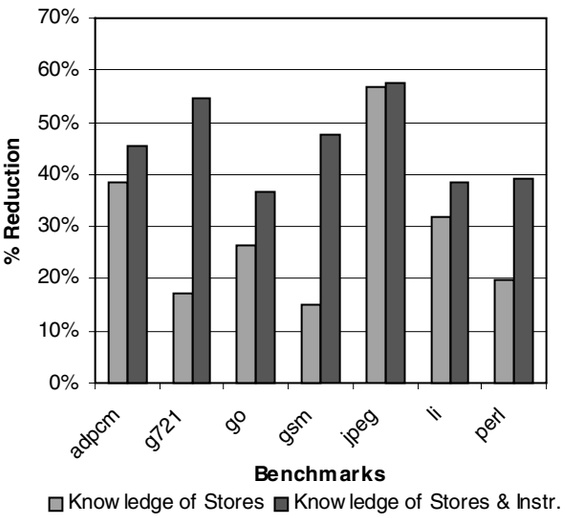


Fig. 9. Percentage decrease in bit flips without knowledge about load values.

The figure also shows the effect of reordering with knowledge about instruction values (with knowledge about stores and execution paths). For instruction reordering, the compiler could arrange instructions in an order that causes the minimal signal transitions. The reordering scope over which the compiler can arrange instructions is limited to sequential code sequences (applying trace-based optimization may help to increase the reordering scope) or to a single cache line. The compiler must convey to the hardware the dependences between instructions that would normally be implicit in sequential code. (Section 5 gives preliminary results for such a scheme, where the compiler reorders instructions within a cache line.)

As the second set of bars in Figure 9 shows, adding knowledge about instruction values to the transaction scheduler greatly improves the ability of the scheduler to reduce switching activity. In this case, the improvement in switching ranges from 37–57%, with an average of 46%. In the figure, *g721* and *gsm* have a large improvement due to knowledge about instruction values. As mentioned previously, for these benchmarks, the switching activity is most strongly associated with the instruction sream. Reordering instructions for these benchmarks greatly improves the switching activity because they are dominated by instruction accesses.

From Figure 9, we conclude that reordering with knowledge about instruction and stored values can reduce switching activity nearly as well as reordering with complete

knowledge about both the instruction and data streams. The reduction shown in Figure 9 compares favorably to the scheme that has full knowledge (see Section 4.1), which reduces switching by an average of 53% (for the HPB and 8K caches).

5 Cache Line Reordering

The previous sections demonstrate that a significant reduction in the switching activity of the memory bus can be obtained by reordering memory bus transactions over a small fixed window of memory accesses. The techniques described above are intended to determine an approximation of the very best that we can achieve given dependences between instruction and data accesses. Based on our encouraging initial results, we are investigating hardware structures to do reordering with compiler support.

Our initial technique, called *cache line reordering* (CLR), considers a small window of accesses. In this scheme, a reordering vector is associated with every cache line that indicates the order in which individual cache line words should be accessed. The memory controller must be able to access memory in an arbitrary order within a cache line and to burst transfer the data to/from the memory system. We reorder accesses within a cache line to find the order that minimizes the transitions for a given line. The CLR scheme has the advantage that it does not have to build a dependence graph dynamically, as the previous technique does.

In this section, we use CLR with a bidirectional bus that has tristate drivers and supports burst transfers. Such a bus is only driven by the CPU or the external memory controller during a cache line transfer. Between transfers, the bus is not driven. Therefore, we do not reorder between cache lines because the switching overhead is effectively always the same when starting a new transfer, regardless of the previous one. However, within a transfer (i.e., cache line), it is possible to reorder elements to minimize switching.

Because a cache line may contain many words (e.g., a 32-byte line has 8 32-bit words), we only reorder within a subline of 4 words. Subline reordering reduces the number of possible combinations of words from *8!* to *2*4!* for a 32-byte line. Although there is a performance loss (i.e., reduction in bit flips) due to reordering over a smaller set than a full cache line, it is more practical to search *4!* combinations than *8!* dynamically in hardware.

CLR is applicable to both data and instructions. The first set of bars in Figure 10 shows the effect of *ideal CLR*. In this case, the hardware always knows the best order in which to read or write every cache line. Unlike the previous experiments, the numbers in the figure were collected using execution-driven simulation that does not have any knowledge about execution paths. The benchmarks in the figure were run to completion, except for *go* and *perl* which were halted after 50 million instructions. The figure shows that CLR can reduce the number of signal transitions by 17–31%, with an average of 25%.

Although ideal CLR is not practical for the data cache (note, however, that we can use profile-feedback optimization to help for data accesses), it is practical for the instruction stream. In this case, the compiler knows at compile time the binary representation of instructions and can reorder instructions within a cache line. To use

this optimization, the compiler needs to know the architecture of the memory-processor interface and to convey to the hardware the reordering vectors associated with every cache line.

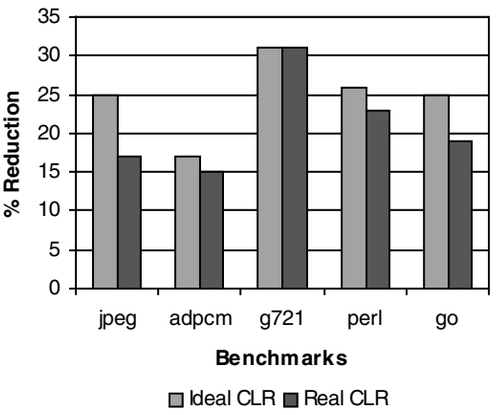


Fig. 10. Percentage reduction in switching activity for CLR. For ideal CLR, the processor always knows the best order in which to read or write cache lines. For real CLR, the hardware reorders instruction cache lines with compiler assistance and reorders data cache lines dynamically.

For the data stream, we can track the best order for each cache line to determine how to transfer it to/from main memory. To do the reordering, the hardware computes the best order for each line whenever it is written back to memory (i.e., a dirty line replacement) or loaded from memory for the first time. The best order for each line is stored in a buffer that is indexed by cache line address. When loading a line, the memory management unit requests the line from main memory in the order specified by the reorder vector in the buffer. The hardware manages the allocation of buffer space.

The second set of bars in Figure 10 (labelled "real CLR") shows the effect of compiler- driven reordering of the instruction stream and hardware dynamic reordering of the data stream. The figure shows that a practical CLR scheme can reduce the overall number of signal transitions significantly. Reordering with a hardware scheme reduces switching activity nearly as well as CLR with perfect knowledge. In this case, the reduction is 15–31% and the average is 21%.

In typical systems, the memory bus can consume a 15–30% of the memory hierarchy's energy [10]. In this case, cache line reordering may reduce energy consumption by 2–9%. Although the reduction is small, we believe that like compiler optimizations, you must apply all the tricks in your bag to substantially reduce overall energy consumption. Cache line reordering is such a technique that can be applied in concert with other energy reduction strategies. Our initial results are promising, and we are continuing to explore techniques for reducing the energy consumption of the

memory hierarchy. A future paper will describe our architecture and implementation of cache line reordering.

6 Related Work

There has been much research on improving power consumption by minimizing processor switching activity. Within a processor, switching has been studied in the context of registers and functional units. Tiwari, Malik, and Wolfe [17,24] perform power-aware instruction selection to reduce the switching effects during execution. Register allocation has been studied with the aim of reducing switching activity [6].

In order to reduce off-chip activity, most previous work focuses on improving cache structures to reduce accesses to external memory. Cache strategies have been analyzed in the context of power consumption [1]. Cooper and Harvey proposed a compiler-controlled on-chip buffer [7] to hold spilled register values to reduce traffic to main memory. Other techniques such as loop [2], filter [15], and victim caches [14] reduce switching by buffering values between processing elements (e.g., between the CPU and on-chip caches or between the CPU and off-chip memory). Our work differs because it aims to reduce switching cost for a given memory hierarchy and cache design.

In [9], the authors reduce the power consumption of the external bus by reordering complex data structures in memory using a cost function. The previous work that relates most to ours is that of [23]. In their work, the switching of off-chip accesses is reduced by encoding addresses with gray codes. Using gray codes, consecutive addresses differ only by a single bit. Gray coding was applied to instruction fetches, which are sequential within a basic block. Our work considers both instruction and data accesses to reduce switching. Also the compiler uses prior knowledge of bus activity during instruction fetches to reorder the instructions.

7 Summary and Future Work

This paper studies reordering bus transactions to reduce the switching activity of the external memory interface. We have developed a tool set called MPOWER for studying the impact of bus structure and transaction reordering on power consumption. In the paper, we use MPOWER to investigate the effectiveness of transaction reordering and in what situations to apply it. We found that for an ideal case, switching activity can be reduced by an average of 53% for a 32-bit non-multiplexed bus. This paper also briefly sketches a feasible hardware scheme for bidirectional buses that can reorder the instruction and data streams with compiler assistance. Our hardware scheme reduces switching activity by 15–31%, leading to a possible energy savings of 2–9%. We are continuing to investigate memory bus transaction reordering. In particular, we are:

- extending MPOWER to determine energy consumption for both memory and the processor;

- developing and implementing a hardware scheme for reordering the elements in a cache line to reduce the energy consumption of the memory bus;
- and developing compiler techniques for reordering the instruction stream with hardware support, and investigating the use of profile-feedback optimizations for reordering the data stream (to reduce hardware complexity).

References

[1] Bahar R. I., Albera G. and Manne S., "Power and performance trade-offs using various Caching Strategies", *Int'l. Symp. on Low-Power Electronics and Design*, pp. 64–69, Monterey, CA, August 1998.

[2] Bellas N., Hajj I., and Polychronopoulos, "Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors", *Int'l. Symp. on Low-Power Electronics and Design*, pp. 70–75, Monterey, CA, August 1998.

[3] Bunda J., Fussell D., Athas W. C., Jenevein R., "16-bit vs. 32-bit instructions for pipelined microprocessors", *ACM/IEEE Int'l. Symp. on Computer Architecture*, pp. 237–246, San Diego, CA, May 1993.

[4] Burger D. and Austin T. M., "The SimpleScalar tool set, version 2.0", Technical Report #1342, Computer Science Department, University of Wisconsin-Madison, June 1997.

[5] Chandrakasan A. and Brodersen R., *Low Power Digital CMOS Design*, Kluwer Academic Publishers, 1995.

[6] Chang J. and Pedram M., "Register allocation and binding for low power", *32nd ACM/ IEEE Design Automation Conference*, pp. 29–35, 1995.

[7] Cooper K. D. and Harvey T. J., "Compiler-controlled memory", 8th ACM Int'l. Conference on Architectural Support for Programming Languages and Operating Systems, pp. 2– 11, October 1998.

[8] Cuppu V., Jacob B., Davis B., and Mudge T., "A performance comparison of contemporary DRAM architectures", *ACM/IEEE Int'l. Symp. on Computer Architecture*, pp. 222– 233, Atlanta, GA, May 1–5, 1999.

[9] Diguet J., Wuytack S., Catthoor F. and Man H. D., "Formalized methodology for data reuse exploration in hierarchical memory mappings", *Int'l. Symp. on Low-Power Electronics and Design*, pp. 30–35, Monterey, CA, August 1997.

[10] Fromm R., Perissakis S., Cardwell N., et al., "The energy efficiency of IRAM architectures", *ACM/IEEE Int'l. Symp. on Computer Architecture*, pp. 327–337, Denver, CO, June 1997.

[11] Gowan M. K., Biro L. L., and Jackson D. B., "Power considerations in the design of the Alpha 21264 microprocessor", *35th ACM/IEEE Design and Automation Conference*, pp. 726–731, San Francisco, CA, June 1998.

[12] Hunt D. and Lesartre G., "PA-8500: The continuing evolution of the PA-8000", *IEEE Computer Conference*, San Jose, CA, February 1997.

[13] Kessler R. E., "The Alpha 21264 microprocessor", *IEEE Micro*, Vol. 19, No. 2, pp. 24– 36, March/April 1999.

[14] Jouppi N., "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers", *ACM/IEEE Int'l. Symp. on Computer Architecture*, pp. 364–373, May 1990.

[15] Kin J., Gupta M., and Mangione-Smith W. H., "The filter cache: A energy efficient memory structure", *IEEE/ACM 30th Int'l. Symp. on Microarchitecture*, pp. 184–193, Research Triangle Park, NC, December 1997

[16] Lee C., Potkonjak M., and Magione-Smith W. M., "MediaBench: A tool for evaluating and synthesizing multimedia and communication systems", *IEEE/ACM 30th Int'l. Symp. on Microarchitecture*, pp. 330–335, RTP, NC, December 1997.

[17] Lee M., Tiwari V., Malik S., and Fujita M., "Power analysis and low-power scheduling techniques for embedded DSP software", *IEEE/ACM Int'l. Symp. on Systems Synthesis*, pp. 110–115, Cannes, France, September 1995.

[18] Lefurgy C., Piccininni E., and Mudge T., "Evaluation of a high performance code compression method", *IEEE/ACM 32nd Int'l. Symp. on Microarchitecture*, pp. 93–102, Haifa, Israel, November 1999.

[19] M•CORE MMC2001 Reference Manual, Motorola Semiconductor Product Sector, Austin, Texas, `www.motorola.com/SPS/MCORE`, 1998.

[20] Motorola PowerPC MPC7400 User's Manual, document number MPC7400UM/D, Motorola Semiconductor Product Sector, Austin, Texas, `www.motorola.com/SPS`, 2000.

[21] Quan M., "Chip makers reap benefits of cell-phone boom", *EE Times*, `www.eetimes.com`, Nov. 18, 1999.

[22] Santhanam S., "StrongARM SA110: A 160 MHz 32b 0.5W CMOS ARM processor", *HotChips VIII*, pp. 119–130, Stanford, CA, August 1996.

[23] Su C-L, Tsui C-Y, and Despain A. M., "Low power architecture design and compilation techniques for high-performance processors", *IEEE Computer Conference*, February 1994.

[24] Tiwari V., Malik S. and Wolfe A., "Compilation techniques for low energy: An overview", *Int'l. Symp. on Low-Power Electronics*, October 1994

A Power Efficient Cache Structure for Embedded Processors Based on the Dual Cache Structure*

Gi-Ho Park[1], Kil-Whan Lee[2], Jae-Hyuk Lee[2],
Tack-Don Han[2], and Shin-Dug Kim[2]

[1] Research Institute of ASIC Design, Yonsei University, Seoul, Korea, 120-749
[2] Department of Computer Science, Yonsei University, Seoul, Korea, 120-749
{ghpark,kiwh,jerry,sdkim,hantack}@kurene.yonsei.ac.kr

Abstract. A dual data cache system structure, called a *cooperative cache system*, is designed as a low power cache structure for embedded processors. The cooperative cache system consists of two caches, i.e., a direct-mapped *temporal oriented cache (TOC)* and a four-way set-associative *spatial oriented cache (SOC)*. These two caches are constructed with different block sizes as well as associativities. The block size of the TOC is 8bytes and that of the SOC is 32bytes, and the capacity of each cache is 8Kbytes. The cooperative cache system achieves improvement in performance and reduces power consumption by virtue of the structural characteristics of the two caches designed inherently to help each other. The cooperative cache system is adopted as the cache structure for the CalmRISC-32 embedded processor that is going to be manufactured by Samsung Electronics Co. with 0.25μm technology.

1 Introduction

Advances in processor technology have considerably elevated the importance of memory hierarchies. Caches as the first level in the memory hierarchy, are designed to utilize fully the locality of reference. Caches function based on the principle of locality. A trade-off between the optimization of spatial and temporal locality transpires in the selection of block size. Increasing the size of a cache block can maximize use of spatial locality, but exploitation of temporal locality will be reduced because the number of resident blocks in the cache will decrease. Several studies have investigated what the optimal size of a block for a given cache capacity should be in view of this effect. Caches constructed with the optimal block size can provide considerable performance, but their capability depends largely on the memory reference pattern of the application programs. Some applications yield good performance for the cache with a specific block size, but other applications may render severely reduced performance with the given block size.

Recent studies try to resolve this drawback by intelligently managing the memory hierarchy that controls the movement and placement of data based on

* This work was supported by Korea Research Foundation Grant for Doctoral Candidates 1998

J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 162–177, 2001.

the type of exploitable locality and data usage characteristics [7,8,9,14,15,17,20]. Studies on dual data cache systems [7,14,17] exploit both spatial and temporal locality by using separate spatial and temporal caches, respectively. Selective caching [7,17,20], cache bypassing [8], and non-temporal streaming cache [15] schemes control the caching and placement of the fetched data based on the existence and amount of exploitable locality. A mechanism intended to detect and optimize spatial locality is presented in [9] to control the amount of data to be fetched adaptively. The above-mentioned cache schemes show that considerable improvement in performance can be achieved over conventional caches based on the individual system's own locality analysis/detection/optimization mechanisms.

This paper presents a cooperative cache system that consists of two caches with different configurations as an alternative approach to designing an power-efficient cache structure with reasonable performance. The cooperative cache system improves the performance of the cache system by virtue of the structural characteristics of the two caches, which are inherently designed to help each other rather than by virtue of any locality prediction/detection/analysis scheme. These two caches are constructed with different associativities as well as block sizes. The cooperative cache consists of a direct-mapped *temporal oriented cache (TOC)* and a highly associative *spatial oriented cache (SOC)*. The size of a block in the TOC is smaller than that of conventional caches, and that of the SOC is equal to or larger than that of conventional caches. The capacity of each cache can differ as well. The cooperative cache system consumes less power based on its structural characteristics, such as small capacity of TOC and accessing mechanism adopted to reduce the power consumption.

Results of performance evaluation show that the cooperative cache system constructed with an 8Kbyte direct-mapped TOC featuring an 8byte block, and an 8Kbyte four-way set-associative SOC with a 32byte block provides 5.3% higher performance on the average (1.4% $\sim$ 15.0%) in CPI than a 16Kbyte conventional cache, and better performance than 64Kbyte direct-mapped cache with 32byte block. Results of the power consumption analysis likewise reveals that the cooperative cache system constructed with the above-mentioned configuration consumes 29.3% less power on the average than direct-mapped 16KByte conventional cache with 32byte block.

Section 2 reviews related work to exploit the locality of reference. The design objectives and structure of the cooperative cache system are explained in Section 3. Section 4 presents simulation results about the performance and the power consumption for both cooperative and conventional cache systems. Section 5 concludes with directions for future research.

2 Related Works

Several studies have focused on intelligent cache management mechanisms that manage memory references differently according to the predicted type of locality [7,8,9,14,15,17,20]. The dual data cache presented by Gonzalez, et al., and

the split temporal/spatial cache system proposed by Milutinovic, et al., are constructed with two separate caches, i.e., spatial and temporal caches, to exploit each type of locality [7,14,17]. These dual cache structures focus on the exploitation of temporal locality, where the temporal cache is specifically designed to exploit temporal locality exclusively with very small cache blocks, i.e., four or eight bytes. The spatial cache, on the other hand, exploits both spatial and temporal locality as conventional caches with the usual block size, e.g., 16 or 32 bytes. These studies, however, do not take into consideration the different associativities between the spatial and temporal caches, unlike the cooperative cache structure, which consists of two caches constructed with different associativities.

Several approaches, such as selective caching [7,17,20], cache bypassing [8], and non-temporal streaming (NTS) cache [15] have been proposed as means to avoid cache pollution by controlling the caching and placement of the fetched data based on the existence and amount of exploitable locality. Selective caching [7,17,20] does not store the fetched data into the cache based on the reuse/locality analysis. Cache bypassing [8] and non-temporal streaming cache [15], on the other hand, store the fetched data into the dedicated buffer, i.e., bypass buffer and non-temporal streaming cache, rather than into the main cache. This transpires when the fetched data are expected to have less locality than the data stored in the main cache or they have no exploitable temporal locality at all.

These previous studies focus on how to detect the type of locality for a certain memory reference and how to deal with the reference based on predicted locality. In contrast, the cooperative cache system presented in this research tries to exploit spatial and temporal locality effectively by adopting a cache configuration designed to reflect the basic property of spatial and temporal locality using the characteristic of the dual cache structure. It also consumes less power based on the dual data cache structure.

3 Cooperative Cache System

A cooperative cache system equipped with two types of data caches constructed with different block sizes and associativities is designed to enhance cache performance. This section presents the design objectives, structure and operational model of the cooperative cache system.

3.1 Design Objectives

This subsection explains two issues that primarily motivated the design of the cooperative cache system. To utilize design flexibility fully is the first motivation that prompted the development of the cooperative cache. Second, the dual cache structure constructed with two separate cache having different configuration is used to reduce the power consumption of cache memory system. The following subsections explain these two motivations in detail.

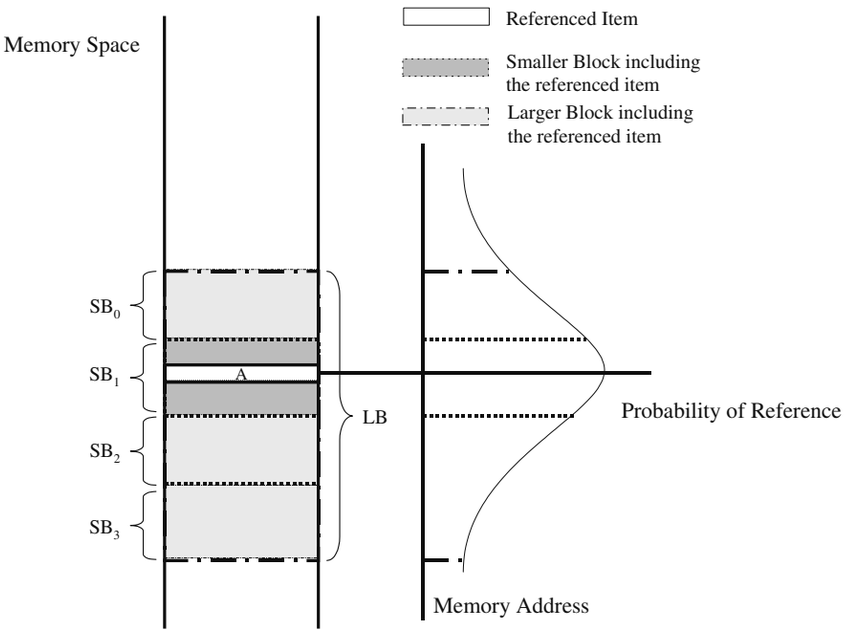


Fig. 1. Relationship between locality and fetch size.

Block Size and Spatial/Temporal Locality. Designing a dual cache structure that can effectively exploit both types of locality for most programs is the primary rationale behind the development of the cooperative cache system. For the purpose of explaining this rationale more comprehensively, the property of locality needs to be clarified first. Throughout this subsection, it is assumed that the fetch size is the same as the block size. Fig. 1 illustrates the relationship between fetch size and the exploitation of both localities when the CPU references a data item (designated as A).

In Fig. 1, SB_1 denotes a small memory block including the data item A, while LB represents a large memory block including the data item A, which is formed as a set of SBs. The size of an LB is assumed to be four times bigger than that of an SB in this example. The graph shown on the right side of the figure presents the probability tendency for any reference with some distance from the referenced item. Although the probability density function in the graph is neither accurate nor realistic, it illustrates the conceptual property of spatial locality that the probability for a data item to be referenced decreases as the distance from the referenced item increases. If an LB is fetched upon a cache miss, given a large fetch size for instance, the miss ratio generally decreases because of the spatial locality of references. However, the expected mean for utilization of a resident cache block is low in fetching LB because the probability for a data item to be referenced decreases as the distance from the referenced item increases. It

would be ideal to design a cache system that can exploit spatial locality fully by fetching a larger memory block, and maximize temporal locality by retaining many smaller memory blocks. However, these optimization goals for both types of locality entail contradictory requirements that the structure of conventional cache systems cannot meet.

The cooperative cache structure presented in this paper satisfies the conflicting requirements previously discussed. The cache system consists of two independent caches: the *spatial-oriented cache (SOC)*, which is constructed with a larger cache block; and the *temporal-oriented cache (TOC)*, which is constructed with a smaller cache block but with numerous cache blocks. When a cache miss occurs, a large SOC block is fetched into the SOC. A TOC block containing the referenced word, e.g., SB1 in Fig. 1, is simultaneously stored into the TOC. Through this operation, the expected mean for utilization of resident blocks in the TOC is kept high while spatial locality is exploited effectively by fetching a large cache block into the SOC. We try to reflect the overall tendency of locality of references by this operation rather than exploit the locality of a specific memory reference based on reuse analysis.

Flexibility in Design of Dual Data Caches. There are many studies on locality optimization techniques [7,8,14,15,17] using the dual cache structure. These studies, however, do not substantially consider the design flexibility provided by two independent caches because their major focus is the effective locality prediction/detection/optimization mechanism.

The dual cache structure constructed with separate spatial and temporal caches [7,14,17] usually concentrates on the exploitation of temporal locality with the use of temporal cache having very small blocks, such as four or eight bytes. Previous studies do not take into consideration the different associativities between spatial and temporal caches. As was explained previously, cache bypassing presented by Johnson, et al., [8] and non-temporal streaming (NTS) cache by Rivers, et al. [15] seek to prevent cache pollution by hindering data expected to have less or no exploitable locality from caching into the main cache. The bypass buffer in [8] stores the data to be referenced less frequently than the data stored in the main cache. The non-temporal streaming cache [15], on the other hand, is constructed as a kind of buffer to store data that are not expected to have temporal locality. Since those buffers are designed to store the data bypassed from the main cache temporarily, their configurations, particularly regarding associativity and capacity, are determined by the function rather than configuration of the main cache. This implies that the effectiveness of those buffers is not closely related to the configuration of the main cache [8]. In summary, the configuration of the dual cache structure in previous studies is closely related to the specific locality prediction, detection, and optimization mechanism presented in each study. This is the main reason why the design flexibility provided by two independent caches has not been fully utilized in those studies.

Hybrid access caches such as victim [10], MRU [5], half and half [19], and column associative cache [1] can also be considered as having a dual cache structure.

Though the associativities of two caches can be different in the hybrid access cache, the block size of the two caches must be the same because data should be exchanged between these two caches. Consequently, hybrid access cache schemes likewise do not fully utilize the design flexibility provided by a dual cache structure. The cooperative cache system, however, best utilizes the design flexibility offered by dual cache systems, including different block sizes, associativities, and capacities to obtain improvement in performance with various cache configurations.

Low Power Comsumption Using Dual Cache Strcuture. Recently, several studies have been proposed for reducing the power consumption of the cache by desining a power efficient cache structure. The filter cache [12] uses a small, i.e., more energy efficient, cache between the CPU and the L1 cache to reduce power dissipated in the cache.

The amount of power consumed by the cooperative cache system is lower than that needed by conventional cache systems by virtue of its small TOC capacity. As in the case of cache access time, The amount of power consumed by the cooperative cache system is lower than that needed by conventional cache systems by virtue of its small TOC capacity. There are a few structural and operational characteristics of the cooperative cache system that reduce power consumption based on the dual cache structure. Those are related with the operational flow of the read/write operation of the cooperative cache system and will be explained later.

3.2 Structure of the Cooperative Cache System

The cooperative cache system consists of two caches, an SOC and a TOC as shown in Fig. 2. Both SOC and TOC are designed in the same way as the structure of conventional caches, but each cache is constructed with its own block size and associativity to enhance the performance of the cache memory. The block size of the SOC is equal to or larger than the best block size of conventional caches whereas that of the TOC is determined to be smaller than the best block size of conventional caches. If the best block size of a conventional cache is 32 bytes for any given cache size, the block sizes of TOC and SOC are usually determined to range from 8 to 16 bytes and from 32 to 64 bytes, respectively. The optimal combination of TOC and SOC block sizes is determined based on the simulation result in the following section. The TOC is constructed as a direct-mapped cache to support faster access time. The associativity of SOC is set high, e.g., four- or eight-way, to alleviate severe conflict misses that can arise from the direct-mapped TOC as well as the SOC itself.

3.3 Operational Flow of the Cooperative Cache System

The *corresponding TOC block* and *corresponding SOC block* are defined to represent a particular TOC block and an SOC block (SB1 and LB1 in Fig. 1) including

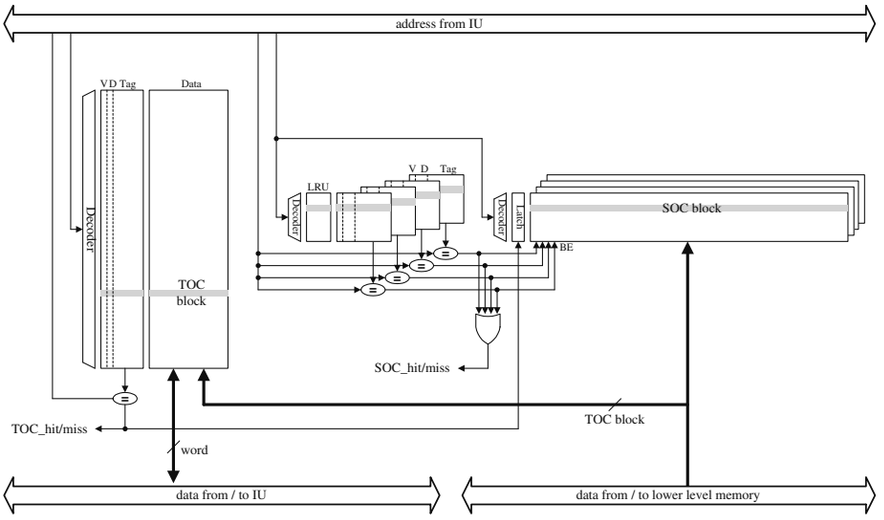


Fig. 2. Structure of cooperative cache system.

a specific data item to be referenced. Detailed discussion on the operations carried out by the cooperative cache system, including read and write operations, are explained in the following subsections.

Read Operations. The cooperative cache performs a read operation in the following manner.

1. In the first cycle, the cache probes a TOC block and K SOC blocks simultaneously. In the case of SOC, the tags of SOC are probed to determine SOC hit or miss.
2. In case of a TOC hit, a read operation completes by simply accessing data from the TOC.
3. If a data item to be read is not residing in the TOC but exists in the SOC, the data item is read from the SOC. In addition to accessing the data item from the SOC, its *corresponding TOC block* is copied into the TOC from the SOC block, i.e., *corresponding SOC block*.
4. When a cache miss occurs, the *corresponding SOC block* is fetched into the SOC from the lower level memory, e.g., the second level cache, or main memory. The *corresponding TOC block* is copied into the TOC as well.

No swapping operation between the TOC and SOC transpires when a TOC block is replaced except when the replaced TOC block is dirty and the SOC block including the TOC block resides in the SOC. The TOC block is written back to the SOC in this case. The dirty TOC block is written-back only to lower level memory in any other cases.

Write Operations. The cooperative cache performs the write function as follows:

1. A hit in the TOC will simply update the contents in the TOC.
2. If there was a miss in the TOC and a hit in the SOC, the *corresponding TOC block* is copied into the TOC from the SOC and modified while the data in the SOC is not modified. Because TOC has a priority over the SOC in a read operation, the write operation for the SOC block can be delayed until the TOC block is replaced.
3. When a TOC block with stale data is replaced from the TOC, the SOC block including the TOC block is updated only if it is resident in the SOC. If the SOC block including the TOC block is not resident in the SOC, the lower level of memory, namely the second level cache or main memory, is updated.

Maintaining Coherency. There may be coherency problems in the cooperative cache system because it has two caches that have the same cache block. Maintaining all the TOC data as up-to-data through the read, and write operations serve as solution to the coherency problems that may be occurred on-chip. The TOC is probed first in the read operation and all write operations are performed in the TOC. If a dirty TOC block is replaced and an SOC block including the replaced TOC block is resident in the SOC.

The multi-level inclusion property [3] can be violated in the cooperative cache system as in other dual cache systems. If it is required to observe multi-level inclusion property in the multiple processor systems, inclusion property is maintained through one of three methods presented by Rivers [16]. Those methods are: making the configuration of the L2 cache (capacity, associativity, and block size) to maintain inclusion property, duplicating L1 tags in L2 directory and duplicating L1 contents into the extra storage at the L2 level.

Two-Level Cache vs. Cooperative Cache Systems. At first glance, the conceptual framework and operational flow of the cooperative cache appear to be similar to those of a two-level cache. This is because both systems have a slower secondary cache, i.e., the second-level cache and SOC, to handle misses from the primary cache, i.e., the first-level cache and the TOC, respectively. The cooperative cache system carries out a read operation in the same manner that two-level cache structures perform it. However, several important distinctions between the cooperative cache and the two-level cache exist.

- **Size of second-level cache and SOC:** In two-level cache systems, the capacity of the second-level cache should be larger than that of the first-level cache. However, an SOC smaller than the TOC can be used in the cooperative cache system.
- **Main Purpose:** The second-level cache primarily functions to reduce *capacity misses* with a much larger capacity than the first-level cache. For the cooperative cache, the main purpose of the SOC is to reduce *conflict misses* and *compulsory misses*, with larger cache blocks and higher associativity.

- **Multilevel inclusion property:** This property which is usually maintained in the two-level cache system, is not maintained in the cooperative cache system. Data stored in the TOC may not be resident in the SOC and vice versa, although some replications of data between TOC and SOC are permitted in the cooperative system.
- **Usability of selective caching and locality optimization technique:** The cooperative cache system is designed to take into consideration locality optimization techniques in order to enhance performance. Consequently, selective caching and locality optimization techniques that only load any data block into one of the two caches are easily adapted to the cooperative cache system. This type of loading techniques is not usually allowed in the two-level cache system.

4 Performance Evaluation

This section presents the results of the performance evaluation carried out on the cooperative cache system. The best configuration, including capacities, block sizes and associativities of TOC and SOC, is determined based on the simulation results in Subsection 4.1. The effectiveness of cooperative cache is compared with other caches in Subsection 4.2. Subsection 4.3 presents the results of the power consumption analysis.

4.1 Configuration of the Cooperative Cache System

The best configuration of the cooperative cache system, i.e., the best combination of TOC and SOC capacities, block sizes, and optimal associativity of SOC, is determined via trace-driven simulation. The variation on the associativity of TOC is not considered because the TOC is basically assumed as a direct-mapped cache in the cooperative cache system.

Benchmarks used in the trace-driven simulation include all of SPECint95 benchmarks. The executables of these benchmarks were processed by QPT2 to generate the traces [2]. Each benchmark runs with train input until its completion except gcc, ijpeg benchmarks. Only data references are collected and used for the simulation. In the cases of gcc and ijpeg benchmarks, an initial total of 150 million data references are collected due to the long execution time. The DineroIV cache simulator is modified to simulate the cooperative cache architecture. The detail results of the performance evaluation to determine the based configuration of the cooperative cache system are leaved out due to the length of paper.

The base cooperative cache systems are chosen as the combination of 8Kbyte direct-mapped TOC having 8 or 16byte cache block and 8Kbyte 4-way set associative SOC having 32 or 64byte cache block, after taking into consideration all the simulation results and the principle design objectives of the cooperative cache system. The performance of those cooperative cache system configurations are evaluated and compared with the conventional caches.

4.2 Performance of Cooperative Cache System

The performance of conventional caches with the same, doubled, and quadrupled capacity are compared with that of the cooperative cache. We performed the execution-driven simulation using the SimpleScalar sim-outorder simulator [4]. The five mediabenchmarks [13] are used for simulation in addition to four SPECint95 and SPECfp95 benchmarks as shown in Table 1. The train inputs are used for SPEC benchmarks until completion except perl, applu, turb3d, wave5, and mpeg2enc. In the case of perl, applu, turb3d, wave5, and mpeg2enc, the simulation is performed for the initial execution of 500 million instructions. The default system configuration of the sim-outorder simulator is used for the simulation. The configuration of the cache system is set as shown in Table 2. Fig. 3 (a),(b) presents the hit ratio and the cycles per instruction (CPI) of the cooperative and conventional cache systems. Various capacities of conventional caches constructed with 16, 32, and 64Kbyte capacity having 16, 32 and 64-byte block are represented as model A to I. The cooperative cache system constructed with 8Kbyte TOC and 8Kbyte SOC are represented by model J and K with the cache block combination of (TOC block, SOC block) as (8, 32) and (16,64) respectively.

Results for model A, C, D, F, G, and I are not presented in Fig. 3 because they exhibit worse performance than other cache configurations with the same capacity. The hit ratio yielded by various caches is shown in Fig. 3 (a). The portion of hits serviced by each cache, i.e., TOC and SOC, in the cooperative cache system are presented. The hit ratio of the primary cache in the model B, E, and H, is higher than that of TOC in model J and K because the size of primary cache in those models is 16, 32 or 64Kbytes while that of TOC in model J and K is 8Kbytes. Fig. 3 (b) presents the performance of various conventional and cooperative caches.

The cooperative cache (model J and K) achieved a performance gain of 5.3% (model J) and 5.0% (model K) on the average over the conventional caches in the point of CPI. The cooperative cache constructed with 8Kbyte SOC and TOC (model J and K) provides better performance than the 32Kbyte and 64Kbyte conventional caches (model E, H) on the average in terms of CPI as shown in Fig. 3 (b). It is revealed that the cooperative cache systems constructed with two caches having different capacities, such as 4Kbyte or 16Kbyte SOC with 8Kbyte TOC also provides comparable performance with the base cooperaitve cache system through the performance evaluation. The performance evaluation results, however, are not presented in this paper due to the length limit.

4.3 Power Consumption of Cooperative Cache System

There are a few structural and operational characteristics of the cooperative cache system that reduce power consumption.

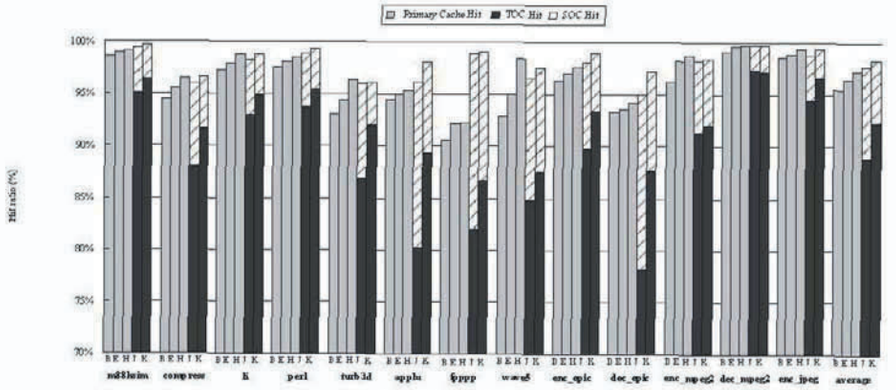
- TOC hit: only a small-sized TOC is accessed.
- SOC hit: only the way (cache array) that contains the data to be accessed is activated. As explained in Section 3.3, This can be available because the

Table 1. Benchmarks used for simulation.

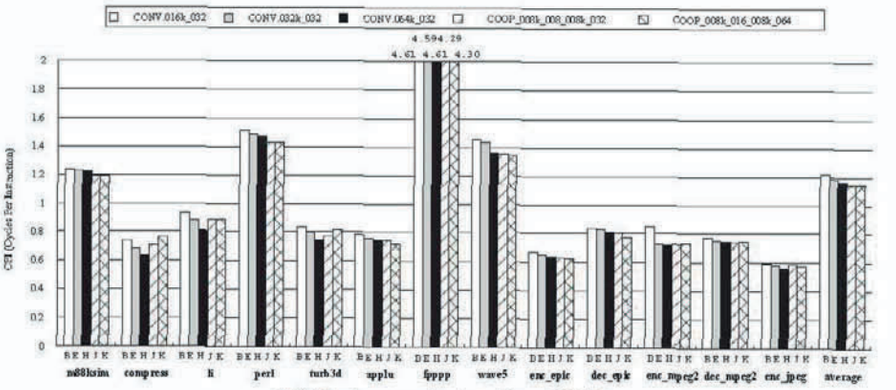
Program		Program Description	Instruction Count (millions)	Memory References (millions)	
				Loads	Stores
SPEC CINT95	m88ksim	A chip simulator for the Motorola 88100 microprocessor	120.1	22.79	14.255
	compress	A in-memory version of the common UNIX utility	35.684	7.366	5.989
	li	Xlisp interpreter	183.282	47.471	30.334
	perl	An interpreter for the Perl language	500	122.792	90.762
SPEC CFP95	turb3d	Turbulence modeling	500	67.278	41.661
	applu	Partial differential equations	500	103.9	23.693
	fpppp	From Gaussian series of quantum chemistry benchmarks	329.849	126.337	48.811
	wave5	Maxwell's equations	500	72.165	37.775
Media-benchmark	epic	Data compression using wavelet decomposition and Huffman coding	52.722	6.765	0.773
	unepic	Epic decoding wavelets and Huffman coding	6.809	0.888	0.757
	mpeg2enc	Encoding in the standard MPEG digital compressed format	500	135.567	11.334
	mpeg2dec	Decoding in the standard MPEG digital compressed format	171.233	26.53	6.354
	cjpeg	Image comperssion using DCT algorithm	15.504	3.251	1.004

Table 2. Cache system models used for simulation.

		Conventional Cache									Cooperative Cache (TOC,SOC)	
Model		A	B	C	D	E	F	G	H	I	J	K
First-Level Data Cache	Cache Size (Kbytes)	16			32			64			(8,8)	(8,8)
	Block Size (Byte)	16	32	64	16	32	64	16	32	64	(8,32)	(16,64)
	Associativity	1									(1,4)	
	Cycle(s) for Access	1									(1,2)	
	Area (mm^2, in 0.25um)[7]	11.38422	9.912965	9.271257	22.53809	19.59517	18.21763	44.72232	38.89828	36.08018	12.48291	11.22015
Fisrt-Level inst. Cache		16KByte, 32byte cache block, direct-mapped, 1 cycle access time										
Second-Level Cache		None										



(a) Hit ratios.



(b) Cycles per instruction (CPI).

Fig. 3. Performance comparisons of cooperative cache and conventional cache.

SOC tag compare is performed one cycle before the SOC access. The tag matching information is used to enable the cache way that contains the data to be accessed.
- SOC miss and TOC miss: only TOC and SOC tags are accessed as well. SOC is not accessed because the SOC cache array is accessed only when the SOC tag compare result is hit.

The following parameters are defined to estimate the power consumption of cooperative cache systems and conventional cache systems.

- C : cache size in Kbytes.
- B : cache block size in bytes.
- A : cache associativity.
- $C(C, B, A)$: cache system that has the configuration of (C, B, A).
- $P_{read}^{C(C,B,A)}$: power consumed by a read operation when the cache configuration is (C, B, A).
- $P_{write}^{C(C,B,A)}$: power consumed by a write operation when the cache configuration is (C, B, A).
- $H(C, B, A)$: hit ratio of the cache that has configuration (C, B, A).
- $H_{TOC}(C, B, A)$: hit ratio of TOC that has configuration (C, B, A).
- $H_{SOC}((C_1, B_1, A_1), (C_2, B_2, A_2))$: hit ratio of SOC that has configuration (C_2, B_2, A_2) with the TOC of (C_1, B_1, A_1).
- $P_{miss}^{C(C,B,A)}$: power consumed by a cache miss with the cache configuration (C, B, A).
- $P_{miss}^{C((C_1,B_1,A_1),(C_2,B_2,A_2))}$: power consumed by a cache miss in the cooperative cache system that is constructed with the (TOC, SOC) cache configuration $(C_1, B_1, A_1), (C_2, B_2, A_2)$.

Power consumed by the cache system is calculated based on the $0.25\mu m$ ASIC Library data book from Samsung Electronics Co. [18]. The power consumed by the main memory access is assumed as 18.5nJ per access (when cache block size is 32bytes) based on [12]. If the cache block size is 64bytes like in model K, the power consumed by the main memory access is assumed as 37nJ per access. Table 3 presents the power consumption of various cache configurations per read and write operation. The average power consumed by a cache access in a cooperative cache system and a conventional cache system is calculated using the parameters as follows.

- Conventional cache with cache configuration (C, B, A):

$$P_{Conv}^{(C,B,A)} = \frac{\#\_of\_load \times P_{read}^{C(C,B,A)} + \#\_of\_store \times P_{write}^{C(C,B,A)}}{\#\_of\_load + \#\_of\_store} + (1 - H(C, B, A)) \times P_{miss}^{C(C,B,A)} \tag{1}$$

- Cooperative cache with cache configuration $((C_1, B_1, A_1), (C_2, B_2, A_2))$ of TOC and SOC combination:

$$P_{Coop}^{((C_1,B_1,A_1),(C_2,B_2,A_2))} = \frac{\#\_of\_load \times P_{read}^{C(C_1,B_1,A_1)} + \#\_of\_store \times P_{write}^{C(C_1,B_1,A_1)}}{\#\_of\_load + \#\_of\_store}$$

Table 3. Power consumption of various cache configuration per read/write operation. (nJ/operation)

Cache Size	Associativity	Block Size	Operation	
			read	write
8K	1	8	0.254	0.552
		16	0.253	0.438
		32	0.350	0.445
	4	32	0.689	1.040
		64	0.720	1.015
16K	1	16	0.272	0.561
		32	0.355	0.484
32K	1	16	0.459	0.621
		32	0.368	0.565

$$+(1 - H_{TOC}(C_1, B_1, A_1)) \times P_{read}^{C(C_2,B_2,A_2)}$$
$$+(1 - H_{TOC}(C_1, B_1, A_1) - H_{SOC}((C_1, B_1, A_1), (C_2, B_2, A_2)))$$
$$\times P_{miss}^{C((C_1,B_1,A_1),(C_2,B_2,A_2))} \quad (2)$$

Some details are left out to simplify the estimation in the above equations. Fig. 4 shows the power consumption pattern of cooperative cache systems and conventional cache systems. The cache system that has a smaller cache block size usually sets up less power. The cooperative cache constructed with (32, 8) of SOC, TOC block combination shows the least power consumption on the average. The cooperative cache system consumes 29.3% less power than the conventional cache system. The cooperative cache system constructed with (32, 8) of SOC, TOC block combination is appropriate as a cache structure for low-power oriented system, such as battery-backup portable devices.

5 Conclusion

The cooperative cache structure represents an innovative approach to cache design that takes advantage of the design flexibility of dual data cache systems. The two caches that comprise the cooperative cache structure are designed in such a way that the TOC is a direct-mapped cache with smaller cache blocks, while SOC is a set-associative cache with larger cache blocks. The proposed cache design which consists of a direct-mapped TOC with 8∼16 byte block and a four-way set-associative SOC with 32∼64 byte block was found to provide considerable improvement in performance. When it is constructed with an 8Kbyte direct-mapped TOC and an 8Kbyte four-way set-associative SOC with (8, 32)

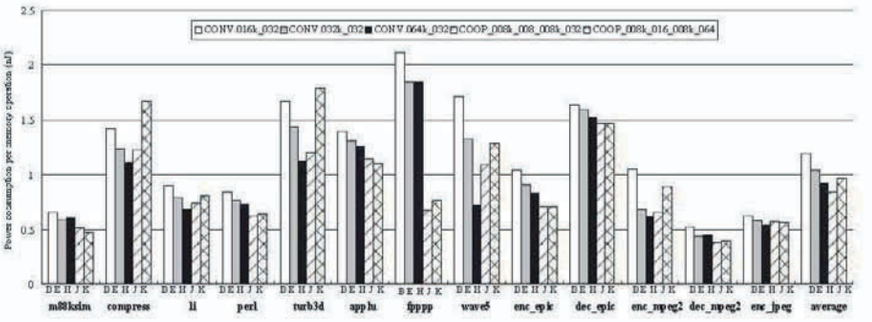


Fig. 4. Power consumption of various cache configuration per memory operation. (nJ/operation)

and (16, 64) of TOC and SOC block combination, the cooperative cache system yielded a better performance than a 64Kbyte conventional cache on the average. Results of the analysis on power consumption reveals that the cooperative cache system is power-efficient. On the average, the cooperative cache system constructed with an 8Kbyte direct-mapped TOC with an 8byte block and an 8Kbyte four-way set-associative SOC with 32byte block consumes less power than the direct-mapped 16KByte conventional cache with 32byte block by 29.3%.

References

1. Anant Agarwal, and Steven D. Pudar, "Column Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches," *20th ISCA*, Vol. 16, No. 4, July 1994, pp. 1319-1360.
2. T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *ACM Trans. on Programming Languages and Systems*, Vol. 16, No. 4, July 1994, pp. 1319-1360.
3. Jean-Loup Baer, and Wen-Han Wang, "On the Inclusion Properties for Multi-Level Cache Hierarchies," *15th ISCA*, May 1988, pp. 73-80.
4. Doug Burger, and Todd Austin, "Evaluating Future Microprocessor: the SimpleScalar Tool Set," *Technical Report #1342*, University of Wisconsin-Madison, June 1997.
5. J. H. Chang, H. Chao, and K. So, "Cache Design of a Sub-Micron CMOS System/370," *14th ISCA*, May 1987, pp. 208-213.
6. Steven Fu, "Cache Design Tool," *available in http://umunhum.stanford.edu /tools/cachetools.html.*
7. A. Gonzalez, C. Aliagas and M. Mateo, "Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality," *Supercomputing '95*, July 1995, pp. 338-347 .
8. Teresa L. Johnson, and Wen-mei W. Hwu "Run-time Adaptive Cache Hierarchy Management via Reference Analysis," *24th ISCA*, June 1997, pp. 315-326.

9. Teresa L. Johnson, Matthew C. Merten, Wen-mei W. Hwu, "Run-time Locality Detection and Optimization," *MICRO-30*, Dec. 1997, pp. 57 - 64.
10. Norman P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffer," *17th ISCA*, May 1990, pp.364-373.
11. Milind B. Kamble, and Kanad Ghose, "Analytical Energy Dissipation Models for Low Power Caches," *ISPLED-97*, Aug. 1997, pp.143-148.
12. Johnson Kin, Munish Gupta, and William H. Mangione-Smith, "The Filter Cache: An Energy Efficiency Memory Structure," *MICRO-30*, Dec. 1997, pp.184-193.
13. Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith, "Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems," *MICRO-30*, Dec. 1997, pp.330-335.
14. V. Milutinovic, M. Tomasevic, B. Markovic, M. Tremblay, " The Split Temporal/Spatial Cache: Initial Performance Analysis," *SCIzzL-5*, March 1996.
15. Jude A. Rivers, and Edward S. Davidson, "Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design," *1996 ICPP*, Aug. 1996, pp. 154-163.
16. Jude A. Rivers, *Performance Aspects of High-Bandwidth Multi-Lateral Cache Organizations,* Ph.D Dissertation, Univ. of Michigan, 1998, pp.30-31.
17. F. Jesus Sanchez, Antonio Gonzalez, and Mateo Valeo, "Static Locality Analysis for Cache Management," *PACT'97*, Nov. 1997, pp. 261-271.
18. Samsung Electronics, *ASIC MDL110 Data Book: 0.25μm 2.5V CMOS Standard Cell Library for Pure Logic/MDL Products,* 1999. pp. 5-32 - 5-44.
19. Kevin B. Theobald, Herbert H. J. Him, Guang R. Cao, "A Design Framework for Hybrid-Access Caches," *HPCA-1*, Jan. 1995, pp. 144-153.
20. Gary Tyson, Matthew Farrens, John Matthews and Andrew R. Plezkun, "A Modified Approach to Data Cache Management," *MICRO-28*, Dec. 1995, pp. 93-103.
21. Steven J. E. Wilton, and Norman P. Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," *IEEE Journal Solid-State Circuits*, Vol. 31, No. 5, May 1996, pp. 677-688.

Approximation of Worst-Case Execution Time for Preemptive Multitasking Systems

Matteo Corti[1], Roberto Brega[2], and Thomas Gross[1]

[1] Departement Informatik, ETH Zürich, CH 8092 Zürich
[2] Institute of Robotics, ETH Zürich, CH 8092 Zürich

Abstract. The control system of many complex mechatronic products requires for each task the Worst Case Execution Time (WCET), which is needed for the scheduler's admission tests and subsequently limits a task's execution time during operation. If a task exceeds the WCET, this situation is detected and either a handler is invoked or control is transferred to a human operator. Such control systems usually support preemptive multitasking, and if an object-oriented programming language (e.g., Java, C++, Oberon) is used, then the system may also provide dynamic loading and unloading of software components (modules). Only modern, state-of-the art microprocessors can provide the necessary compute cycles, but this combination of features (preemption, dynamic un/loading of modules, advanced processors) creates unique challenges when estimating the WCET. Preemption makes it difficult to take the state of the caches and pipelines into account when determining the WCET, yet for modern processors, a WCET based on worst-case assumptions about caches and pipelines is too large to be useful, especially for big and complex real-time products. Since modules can be loaded and unloaded, each task must be analyzed in isolation, without explicit reference to other tasks that may execute concurrently.
To obtain a realistic estimate of a task's execution time, we use static analysis of the source code combined with information about the task's runtime behavior. Runtime information is gathered by the performance monitor that is included in the processor's hardware implementation. Our predictor is able to compute a good estimation of the WCET even for complex tasks that contain a lot of dynamic cache usage, and its requirements are met by today's performance monitoring hardware. The paper includes data to evaluate the effectiveness of the proposed technique for a number of robotics control kernels that are written in an object-oriented programming language and execute on a PowerPC 604e-based system.

1 Introduction

Many complex mechatronic systems require sophisticated control systems that are implemented in software. The core of such a system is a real-time scheduler that determines which task is granted access to a processor. In addition, such systems may include a (graphical) user interface, support for module linking (so that a family of robots can be handled), a facility to dynamically load or unload modules (to allow upgrades in the field), and maybe even a network interface (to retrieve software upgrades or allow remote diagnostics). Such systems are not autonomous, i.e., there is exists a well-defined re-

J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 178–198, 2001.

sponse to unexpected events or break-downs, and such a response may involve a human operator.

The core system shares many features with a real-time system, and as in a real-time system, each task has a computational deadline associated with it that must be met by the scheduler. To allow the scheduler admission testing, it needs to know the Worst Case Execution Time (WCET) for each task. This WCET, or the maximum duration of a task, is of great importance for the construction and validation of real-time systems. Various scheduling techniques have been proposed to enforce the guarantee that tasks finish before their deadlines, but these scheduling algorithms generally require that the WCET of each real-time task in the system is known a priori [4].

Modern processors include several features that make it difficult to compute the WCET at compile time, such as out-of-order execution, multiple levels of caches, pipelining, superscalarity, and dynamic branch prediction. However, these features contribute significantly to the performance of a modern processor, so ignoring them is not an option. If we make always worst-case assumptions on the execution time of an instruction, then we overestimate the predicted WCET by an order of magnitude and consequently obtain only a poor usage of the processor resources.

The presence of preemptive multitasking in the real-time system worsens this overestimation because the effects of a context switch on caches and pipelines cannot be easily foreseen. The difficulties to analytically model all aspects of program execution on a modern microprocessor are evident if we consider the requirements of controllers for modern mechatronic systems. In addition to real-time tasks, the controller must manage a dynamic set of other processes to handle computations that are not time-critical as well as input/output, including network and user interaction.

In such a dynamically changing system, the constellation of real-time and other tasks cannot be predicted. Since all these tasks occupy the same host together, the effects of preemption are difficult, if not impossible, to predict. And a purely analytical off-line solution to the problem is not practical.

Our approach to this problem consists of two parts: We determine an approximation of the WCET for each task. Then the control systems decides, based on the WCET, if the task with its deadline can be admitted, and if admitted, monitors its behavior at runtime. Since the WCET is only an estimate (although a good one, as we demonstrate in this paper), the scheduler checks at runtime if a task exceeds its estimated WCET. A basic assumption is that the system can safely handle such violations: The scheduler attempts to meet all deadlines, but if during execution a deadline is missed since a task takes longer than estimated, then there exists a safe way to handle this situation. This assumption is met by many industrial robot applications. In the absence of a well-founded WCET estimate, the system must be prepared to deal with the developer's *guess* of the WCET.

To determine a realistic estimate of the WCET, our toolset integrates source code analysis of statistical dynamic performance data, which are gathered from a set of representative program executions. To collect runtime information, we use a processor-embedded performance monitor. Such monitors are now present in many popular architectures (e.g., Motorola PowerPC, Compaq Alpha, and Intel Pentium).

Although this approach does not establish a strong bound on the execution time (cost-effective performance monitors provide only statistical information about the execution time), we obtain nevertheless an estimate that is sufficiently accurate to be of use to a real scheduler.

2 Problem Statement

The objective is to develop a tool to determine a realistic approximation of the maximum duration of a given task on a modern pipelined and superscalar processor. The system that executes the task is controlled by a real-time operating system that includes preemptive multitasking. Yet an unknown and dynamic set of other processes that run concurrently with the real-time tasks prevents an off-line a priori scheduling analysis. The state of the processor and the caches is unknown after a context switch. We assume that the execution platform (or an equivalent system) as well representative input vectors are available for measurements.

The computation of the approximate WCET can involve the compiler and, if necessary, the user, but the structure of the tool must be simple and suitable for use by application experts, not computer scientists. Therefore interaction with the tool by the program developer should be minimized.

To support modern program development techniques, we want to be able to handle programs written in object-oriented languages. Such programs have a number of properties that make integration into a real-time environment difficult. But the use of such a language is a requirement imposed by the application developers [2]. Since the computation of a task's maximum duration is in general undecidable, we impose restrictions in the programs that are considered: For those loops and method invocations for which a compiler cannot determine the trip count or call chain, the user must provide a hint. (Recall that the real-time system checks at runtime if a task exceeds its computed maximum duration. The consequences of incorrect hints are detected.) Furthermore, we do not permit the use of the `new` construct in a real-time task, since this operation is not bounded in time or space. These restrictions are accepted by our user community that wants to take advantage of the benefits of an object-oriented programming style but can comfortly live without storage allocation in a real-time controller.

Our paper presents a solution to this problem that handles programs written in Oberon-2 [24], a Pascal-like language with extensions for object-orientation. The solution presented addresses the problems that are faced by users of similar languages, e.g., Java.

3 System Overview

The tool for the WCET approximation is implemented as part of the XOberon preemptive real-time operating system and compiler developed at the Institute of Robotics, Swiss Federal Institute of Technology (ETH Zürich) [2,6]. The execution platform is a set of Motorola VME boards with a PowerPC 604e processor [12]. We summarize here a few aspects of this platform. XOberon is loosely based on the Oberon System [33] and

includes a number of features to support the rapid development of complex real-time applications; e.g., the system is highly modular (so different configurations can be created for specific development scenarios), and the system recognizes module interfaces that are checked for interface compatibility by the dynamic loader.

XOberon includes a deadline-driven scheduler with admission testing. The user must provide the duration and the deadline of a task when submitting it to the system. The real-time scheduler preallocates processor time as specified by the duration/deadline ratio. If the sum of all these ratios remains under 1.0, the scheduler accepts the task, otherwise the task is rejected. The scheduler is also responsible for dispatching non–real-time tasks, which are brought to the foreground only when no other real-time task is pending or waits for dispatch.

The system handles non-compliance by stopping any task that exceeds its time allotment. Such a task is removed from the scheduling pool, and the system invokes an appropriate exception handler or informs the operator. This behavior is acceptable for applications that are not life-critical and that allow stopping the complete system. The use of interrupts is deprecated, and the system and applications predominantly use polling to avoid unexpected interruptions during a task's execution.

WCET prediction proceeds as follows: The source of a real-time task is first compiled, and the resulting object code is run on the target platform with some training input. The execution of this real-time task is monitored in the actual system with the same set of processes that will be present in the final system. Thereafter, our tool uses the statistics from the training runs, as well as the source code, to approximate the WCET.

4 Related Work

The importance of WCET computation has rapidly grown in the past years and attracted many research groups [30]. These efforts differ in the programming language that is supported, the restrictions that are imposed on programs admitted for real-time scheduling, the (real-time) runtime system that controls the execution, and the hardware platforms that are used. We first discuss programming language aspects and then consider different approaches to deal with modern processors.

Kligerman and Stoyenko [15], as well as Puschner and Koza [28], provide a detailed description of the constraints that must be obeyed by the real-time tasks. The requirements include bounds on all loops, bounded depth of recursive procedure calls, and the absence of dynamic structures. To enforce these restrictions they introduce special language constructs into the programming language, Real-Time Euclid and MARS-C, respectively. Other groups use separate and external annotations and thereby decouple the programming language from the timing specifications [23,27].

We choose to embed information about the real-time restrictions—i.e., the maximum number of loop iterations and the cycle counts of any input/output—in the program source and to integrate our predictor with the compiler. This approach gives the predictor a perfect knowledge of the mapping between the source code and the object code produced, even when some simple optimizations take place.

To avoid a lot of annotations in the source file, we perform a loop termination analysis that computes the number of iterations for many loops. This compiler analysis phase simplifies the work of the user and reduces the necessity of specific tools for the constraint specification [16]. The implementation of this optimization is greatly eased by the absence of pointer arithmetic and the strong typing embodied in the programming language (Oberon-2 in our system, but the same holds true for other modern languages, e.g., Java).

The second aspect of computing the WCET is dealing with the architecture of the execution platform. The first approaches in this field were based on simple machines without caches or pipelines. Such machine models allow a precise prediction of the instruction lengths, because the execution time of a given instruction is always known. However, the rising complexity of processor implementations and memory hierarchies introduces new problems to deal with. The execution time of instructions is no longer constant; instead it depends on the state of the pipelines and caches. Several research groups have investigated worst-case execution time analysis for modern processors and architectures.

For fixed pipelines, the CPU pipeline analysis is relatively easy to perform because the various dependences are data-independent. Several research groups combine the pipeline status with the data-flow analysis to determine the WCET [8,19,22,34]; this approach can be extended to the analysis of the instruction cache, since the behavior of this unit is also independent of the data values [8,19,32].

Data caches are a second element of modern architectures that can severely influence the execution time of load and store instructions. To precisely analyze the behavior of the data caches we need information about which addresses are accessed by memory references. If we treat each access as a cache miss, then the predicted time is tremendously overestimated (given the differences between the access time to cache and memory). We can attempt to reduce this overestimation with data-flow analysis techniques [13,19] relying on control of compiler or hardware. The results reported highly depend on the examined program and on the number of dynamic memory accesses that are performed; as the program's complexity grows, the predictions and actual execution times differ significantly.

Another approach to the data cache problem focuses on active management of the cache. If we can prevent the invalidation of the cache content, then execution time is predictable. One option is to partition the cache [14,20] so that one or more partitions of the cache are dedicated to each real-time task. This approach helps the analysis of the cache access time since the influence of multitasking is eliminated, but on the other hand, partitioning limits the system's performance [21].

Techniques exist to take into account the effects of fixed-priority preemptive scheduling on caches [17,3], but the nature and flexibility of the process management of XOberon (see Sect. 3) prevents an off-line analysis with acceptable results.

All these approaches try to reduce the WCET solely by static analysis. The results vary greatly depending on the examined program: If the number of dynamic memory operations is high, such as in linear algebra computations that involve matrices, the prediction may significantly exceed the real maximum duration.

Good results can be obtained for straight-line code [31] (i.e. code without loops and recursive procedure calls), but problems arise when, such as in our case, the system provides preemptive multitasking and does not enforce special restriction on the real-time task structure. The program flow can be interrupted at any time, flushing the pipeline and forcing caches to be refilled. Such dynamic behavior cannot be handled by pure static analysis, because the resulting worst-case execution time would be too large to have any practical relevance. Out-of-order execution and branch prediction are other characteristics that increase the number of variables that change dynamically depending on the input data.

The novelty in our approach consists in the use of a runtime performance monitor to overcome the lack of information caused by preemption and by the dynamic nature of the XOberon system. We monitor the use of pipelines and caches at runtime, and we feed these data in the predictor to approximate the cycle counts of each instruction. This information is then combined with the insight obtained from static analysis to generate the final object code for the real-time task. Multitasking effects, like cache invalidations and pipeline flushes, are accounted by the gathered data. This approach not only minimizes the impact of preemption but also helps us to deal with dynamic caching, out-of-order execution, and speculative execution in a simple, integrated, and effective way.

5 Static Analysis of the Longest Path

To determine the WCET of a task, we compute the longest path on a directed and weighted graph representation of the program's data flow. The nodes of the graph represent the basic blocks, while the edges represent the feasible paths. Each edge is weighted with the length (number of cycles) of the previous basic block.

The WCET corresponds to the longest path of the graph. This path is obviously computable only if the maximum number of iterations of each loop is known and there are no direct or indirect recursive method invocations. This technique results in an overestimation of the WCET, since mutually exclusive control flow branches are ignored. The false path problem is in general undecidable, but heuristics to deal with the problem have been proposed [1].

To keep the user involvement in computing the WCET as low as possible, the WCET estimator analyzes the source code trying to automatically bound the maximum number of loop iterations. The user needs only to specify the cost (in number of cycles) of the memory mapped input/output operations as well as the loop bound for those loops, for which the tool is unable to compute a limit on the number of iterations. This situation can occur when the loop termination condition depends on input values. For the majority of the programs, however, the compilation and analysis pass is successful in determining the maximum number of iterations.

Although our analysis is simple (many compilers employ more sophisticated techniques to identify induction variables [25]), we obtain good results because the majority of the loops in real-time software have a fixed length and a for-like structure. In our suite of test programs (see Sect. 8) only a few loops needed a hint in the form of a bound specification.

Our tool reports loops that defy analysis; for those loops, a hint is required specifying the maximal number of iterations. Such hints are syntactic comments, and the modified programs can still be compiled with any Oberon-2 compiler.

When the user manually specifies a loop bound, the compiler generates code to check at runtime that the hint is correct. If the loop bound is exceeded during the execution, an exception is raised and the program is halted, helping the developer to identify a misleading hint.

Although our simple solution gives very good results, as the vast majority of the loops in the tested programs was automatically bounded, many improvements are possible. There exist algorithms to deal with loops with multiple exits; other researchers have developed solutions to compute the trip count for loops with multiple induction variables or to predict the execution time of loops where the number of iterations depends on counter variables of outer-level loops [9,10].

Object orientation may cause problems when determining the WCET, because at compile time, the type of a given object and the dynamic cycle costs of its methods are not known. This problem is especially annoying when we consider invocation of I/O routines, which involve the runtime system and are beyond the scope of the compiler. The XOberon system uses a database of objects, retrieved by name, to store the different drivers. We introduce a new construct to allow the programmer to insert a hint that expresses the cost of a driver method call (measured in processor cycles).

When each loop in the graph representation of the task has been bounded, we sum the cycle counts of all instructions in each basic block (details in Sect. 6) and record this sum as the weight of the outgoing edges of each node. To compute the longest path on the graph, this graph must be transformed into a directed acyclic graph (DAG) by unrolling the loops: The back-edge of the loop is removed, and the weights of the internal edges are adjusted to take the number of repetitions into account. This loop unrolling is done only for this computation and has no effect on the generated code.

6 Instruction Length Computation

Once the number of iterations of each basic block is known, the dynamic length of each instruction in the block is computed. Since the system structure obscures a precise knowledge of the cache and pipeline states, the length of each instruction is approximated using run-time information gathered with a hardware performance monitor. This mapping of monitor information into execution costs is described in the following sections.

6.1 Runtime Performance Monitor

The Motorola PowerPC 604e microprocessor provides hardware assists to monitor and count predefined events such as cache misses, mispredicted branches, and issued instructions [12,18]. Because the scheduler switches a processor's execution among multiple processes, and because statistics about a particular process may be of interest, a process can be marked for runtime profiling. This feature helps us to gather data only on the task of interest and avoids interference from other processes.

The performance monitor uses four special registers to keep statistics on the specified events. These registers are read each time the scheduler is activated to avoid additional interrupts. The overhead of the performance monitor routine in the scheduler is small (approximately 30 processor cycles). Monitoring is under user control and therefore does not effect system performance, unless the user wants to gather performance data.

The performance monitor was not specifically designed to help in program analysis but to better understand processor performance. Event counting is not precise, and events are not correctly attributed to the corresponding instructions; this is a common problem for out-of-order execution architectures [5], and the monitor of the PowerPC 604e exhibits this problem. For a given event, we can therefore keep just summary data over a large number of instruction executions. This information is sufficient to evaluate how the processor performs but is too coarse to precisely understand the correlation between a particular event and the instruction generating it.

Many events are not disjoint, and the performance monitor does not report their intersection. Consider, e.g., stalls in the dispatch unit: Seven different stall types are reported, but there is no way to know how many apply to the same instruction. Other problems arise when trying to monitor the number of stalls in the different execution units. The performance monitor reports the number of instruction dependences present in the unit's corresponding reservation station. Unfortunately a dependence of one cycle before the execution does not necessarily imply a stall, because the dependence could be resolved in the meantime. In addition, the PowerPC 604e has split the branch prediction unit (BPU) included in the design of its predecessor (PowerPC 604) into two such units and includes a condition register unit (CRU); see Fig. 1. However the performance monitor events were not updated consistently. The CRU shares the dispatch unit with the BPU, and these two units are treated as a single unit by the performance monitor.

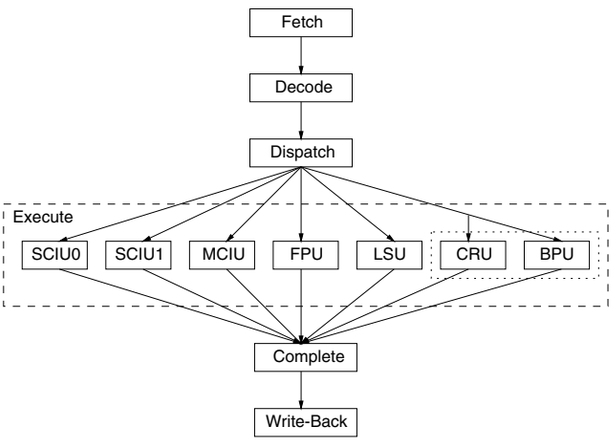


Fig. 1. 604e pipeline structure.

Our WCET approximator deals with these inaccuracies of the performance monitor and tries to extrapolate useful data from information delivered by the hardware monitor to approximate the time it takes to execute each basic block.

The performance monitor is able to deliver information about the cache misses ($miss_{load}$, $miss_{store}$), the penalty for a cache miss ($penalty_{load}$), the idle and busy cycles for each execution unit ($idle_{unit}$, $exec_{unit}$), the load of each execution unit ($load_{unit}$), the dependences in the execution unit's reservation stations (p_{dep}), and the stalls in the dispatch unit (p_{event}).

The length of each basic block (in cycles) is computed by summing up the lengths of the instructions in the block. The length of each block is then multiplied with the number of times the block is executed (see Section 5). We now discuss how the instruction lengths are approximated with help of run-time information.

6.2 CPI Metric

The simplest way to deal with processor performance is to use the popular *instruction per cycle* (IPC) metric, but IPC is a poor metric for our scenario, because it does not provide any intuition about what parts contribute to (the lack of) performance. The inverse of the IPC metric is the *cycles per instruction* (CPI) metric, i.e. the mean dynamic instruction length. The advantage of the CPI metric over the IPC metric is that the former can be divided into its major components to achieve a better granularity [7].

The total CPI for a given architecture is the sum of an infinite-cache performance and the finite-cache effect (FCE). The infinite-cache performance is the CPI for the core processor under the assumption that no cache misses occur, and the FCE accounts for the effects of the memory hierarchy:

$$FCE = (\text{cycles per miss}) \cdot (\text{misses per instruction}) \tag{1}$$

The misses-per-instruction component is commonly called the miss rate, and the cycles-per-miss component is called the miss penalty. The structure of the PowerPC performance monitor forces us to include data cache influences only in the FCE. The instruction cache miss penalty is considered as part of the instruction unit fetch stall cycles (see Sect. 6.4).

The FCE separation is not enough to obtain a good grasp of the processor usage by the different instructions, and we further divide the infinite-cache performance in busy and stall time:

$$CPI = \underbrace{busy + stall}_{\text{Infinite cache performance}} + FCE \tag{2}$$

where the $stall$ component corresponds to the time the instruction is blocked in the pipeline due to some hazard. The PowerPC processor has a superscalar pipeline and may issue multiple instructions of different types simultaneously. For this reason we differentiate between the stall and busy component for each execution unit; let $parallelism$ be the number of instructions that are simultaneously in the respective execution units, $stall_{unit}$ be the number of cycles the instruction stalls in the execution unit, $stall_{pipeline}$

be the number of cycles the instruction stalls in the other pipeline stages, and $exec_{unit}$ the number of cycles spent in an execution unit.

$$CPI = \frac{exec_{unit} + stall_{unit}}{parallelism} + stall_{pipeline} + FCE \tag{3}$$

For each instruction of a basic block we use (3) to approximate the instruction's dynamic length. The difficulty lies in obtaining a correct mapping between the static description of the processor and the dynamic information gathered with the runtime performance monitor to compute the components of (3). The details are explained in the following paragraphs.

6.3 Finite Cache Effect

The PowerPC 604e performance monitor delivers precise information about the mean number of load and store misses and the length of a load miss. Because of the similarity of the load and store operations, we can work with the same miss penalty [12] and compute the FCE as follows:

$$FCE = (miss_{load} + miss_{store}) \cdot penalty_{load} \tag{4}$$

6.4 Pipeline and Unit Stalls

Unfortunately, the performance monitor does not provide precise information about the stalls in the different execution units and the other pipeline stages. The performance monitor was not designed to be used for program analysis, and the predictor must map the data gathered to something useful for the WCET computation.

The first problem arises in the computation of the mean stall time in the different execution units. As previously stated, the performance monitor delivers the mean number of instruction dependences that occur in the reservation stations of the execution units. This number is not quite useful, because a dependence does not necessarily cause a stall, and we must find other ways to compute or approximate the impact of stalls.

The mean stall time for instructions executed by a single-cycle execution unit is easily computable as the difference between the time an instruction remains in the execution unit and its normal execution length (i.e. 1). Let $idle_{unit}$ be the average idle time in a cycle (for an execution unit), and $load_{unit}$ be the average number of instructions handled in a cycle (at most 1):

$$stall_{unit} = \frac{1 - idle_{unit}}{load_{unit}} - 1 \tag{5}$$

Unfortunately the execution time of an instruction in a multiple-cycle execution unit is not constant, and (5) cannot be used. Thanks to Little's theorem applied to M/M/1 queues we can compute the mean length of the reservation station queue N_q (at most 2) and then approximate the mean stall time $stall_{unit}$ as follows (let $\rho = 1 - idle_{unit}$

be the unit's utilization factor (arrival/service rate ratio), and p_{dep} the reported number (probability) of dependences):

$$N_q = \max\left(\frac{\rho^2}{1-\rho}, 2\right) \tag{6}$$

$$(1 - stall_{unit} \cdot load_{unit})^{N_q} = 1 - p_{dep} \tag{7}$$

$$stall_{unit} = \frac{1 - \sqrt[N_q]{1 - p_{dep}}}{load_{unit}} \tag{8}$$

We assume that an absence of reported dependences corresponds to the absence of stalls in the execution unit (7).

A similar problem is present in the computation of the mean number of stall cycles in the dispatch unit: Seven different types of stall are reported for the whole four-instruction queue. We have no way to know how many instructions generated stalls, or how many different stalls are caused by the same instruction. These numbers, however, are important for the understanding of the dynamic instructions lengths, because they include the effects of instruction cache misses and influence all the processed instructions. Several experiments with a set of well-understood test programs revealed that a reported stall can be considered as generated by a single instruction. With p_{event} the reported number of stalls of a given type and $\prod_{event}(1 - p_{event})$ the probability no instruction stalls, we can compute the probability that an instruction stalls:

$$stall_{dispatch} = \frac{CPI \cdot (1 - \prod_{event}(1 - p_{event}))}{queue\_size} \tag{9}$$

The stalls in the dispatch unit ($stall_{dispatch}$) also include stalls in the fetch stage and can be substituted for $stall_{pipeline}$ in (3).

6.5 Execution Length

The normal execution length (i.e. the execution time without stalls) of an instruction for single-cycle units is known from the processor specifications, but the multiple cycle units (FPU, MCIU, and LSU: Floating Point Unit, Multiple Cycle Integer Unit, and Load Store Unit) have internal pipelines that cause different throughput values depending on the instruction type. As a consequence there is no way to know the length of a given instruction, even if we do not consider the caches and pipeline stalls. The presence of preemption, and consequently the lack of knowledge about the pipeline status, forces us to approximate this value.

The architecture specifies the time to execute an instruction ($latency$) and the rate at which pipelined instructions are processed ($throughput$). We consider an instruction pipelined, in the execution unit, if there is an instruction of the same type in the last d_{unit} instructions ($exec_{unit} = throughput$), otherwise its length corresponds to the integral computation time ($exec_{unit} = latency$); the value of d_{unit} is computed experimentally and varies—depending on the unit—from four to eight instructions.

Since our instruction-length analysis is done on a basic block basis, the approximation for the first instruction of a given type in a block is computed differently:

$$p_{instruction} = 1 - (1 - load_{unit})^{d_{unit}} \tag{10}$$

$$exec_{unit} = \begin{cases} latency & \text{if } p_{instruction} < threshold \\ throughput & \text{if } p_{instruction} \geq threshold \end{cases} \tag{11}$$

Where $p_{instruction}$ is the probability that a given instruction type is present in the last d_{unit} instructions.

To check how this approximation works in practice and to refine the d_{unit} and $threshold$ values, we compared for some test programs known mean instruction lengths (obtained with the runtime performance monitor) with the predicted ones, confirming the soundness of the method.

6.6 Instruction Parallelism

We compute the mean instruction parallelism $parallelism$ by considering mean values over the whole task execution, due to the impossibility to get finer grained information. Furthermore, the task cannot be interrupted to check the performance monitor; otherwise interference in the task's flow would destroy the validity of the gathered data. The FCE is taken into account only for the instructions handled by the load/store unit ($util_{LSU}$). Equation 3 can be used again with the mean execution time ($exec$) and mean CPI:

$$parallelism = \frac{exec + stall_{unit}}{CPI - stall_{dispatch} - util_{LSU} \cdot FCE} \tag{12}$$

To compute the mean instruction's execution length $exec$ (needed by (12)), the mean stall value computed earlier can be used. We first compute the mean execution time of the instructions handled by a given execution unit ($exec_{unit}$)—in the same way as the stalls are approximated for the single cycle units.

$$exec_{unit} = \begin{cases} 1 & \text{single cycle} \\ \frac{1 - idle_{unit}}{load_{unit}} - stall_{unit} & \text{multiple cycle} \end{cases} \tag{13}$$

Finally, we compute the total mean instruction length ($exec$) as the weighted mean of all the instruction classes:

$$exec = \sum_{unit} (util_{unit} \cdot exec_{unit}) \tag{14}$$

We have now all the elements ($exec_{unit}$, $stall_{unit}$, $parallelism$, $stall_{pipeline}$ and FCE) needed to compute the length of each instruction of a basic block by applying (3).

6.7 Discussion

We presented an overview of the approximations used to integrate information obtained from the runtime performance monitor with the computation of the dynamic instruction length. The use of probabilistic information and queuing theory does not guarantee exact results, but the presence of preemption naturally prevents a precise analysis, so an approximation is acceptable in this environment. The approximations described here were refined by checking the results of several experimental test programs what are designed to stress different peculiarities of the processor.

This approach could be improved using better suited hardware for the runtime monitor, but as the results (Sect. 8) indicate, the PowerPC 604e performance monitor is sufficient to approximate the WCET with a reasonable precision.

7 Example

The following simple example shows how the gathered information is used to compute the length of the different instructions:

```
1        li r6,996
2        li r5,1000
3        addi r4,r4,-50

4 .LOOP: ldw r3,r4(r5) // the loop is executed 1000 times
5        addi r4,r4,4
6        muli r3,r3,2
7        stw r3,r4(r6)
8        cmpi r4,1000
9        bc .LOOP
```

Table 1. Sample data gathered by the runtime performance monitor

Unit	$load$	$utilization$	$idle$	ρ	N_q	p_{dep}
SCIU0	0.5	0.25	0.4	-	-	-
SCIU1	0.5	0.25	0.4	-	-	-
BPU/CRU	1.0	0.20	0.0	-	-	-
MCIU	0.3	0.10	0.0	1.0	2.0	0.19
FPU	0.0	0.00	1.0	0.0	0.0	0.00
LSU	0.4	0.20	0.0	1.0	2.0	0.23

Table 1 shows some sample data returned by the performance monitor for this code snippet. The impossibility to insert breakpoints in the code and get fine-grained information with the performance monitor forces us to compute global mean values for the stalls and the parallelism. Using the formulae presented in Sect. 6, we are then able to approximate

all the missing values ($stall_{unit}$, $parallelism$, and $stall_{pipeline}$).

$$stall_{SCIU} = \frac{1 - idle_{SCIU}}{load_{SCIU}} - 1 = 0.2 \text{ cycles}$$

In the same way we compute the $stall_{unit}$ for each execution unit (see Table 2) getting

Table 2. Stalls and mean execution times

Unit	$stall$	$exec$	$utilization$
SCIU0	0.20	1.0	0.25
SCIU1	0.20	1.0	0.25
BPU/CRU	0.00	1.0	0.20
MCIU	0.33	3.0	0.10
FPU	0.00	0.0	0.00
LSU	0.35	2.5	0.20
weighted mean	0.20	1.5	

a parallelism of 2.88 and pipeline stall time of 0.09 cycles (the performance monitor reports overall CPI of 0.5 and the effects of data cache as 0.001 cycles per instruction).

$$\begin{aligned} parallelism &= \frac{exec + stall_{unit}}{CPI - stall_{dispatch} - util_{LSU} \cdot FCE} \\ &= \frac{1.5 + 0.2}{0.5 + 0.09 + 0.2 \cdot 0.001} = 2.88 \end{aligned}$$

These values are used for both `addi` instructions (lines 3 and 5), since we are not able to precisely assign the stalls to the correct instruction. We estimate the existence of a stall for the first `addic` instruction (line 3) but since this instruction is executed only once, the effect on the global WCET is not significant.

$$\begin{aligned} CPI_{addic} &= \frac{exec_{addic} + stall_{SCIU}}{parallelism} + stall_{pipeline} \\ &= \frac{1 + 0.2}{2.88} + 0.09 = 0.51 \text{ cycles} \end{aligned}$$

This small example shows how we assign stalls to the different instructions and how the precision of the performance monitor influences the results: The smaller the profiled code segments, the more accurate is the WCET estimation. The global homogeneity and simplicity of real-time tasks permits the computation of a good approximation of the WCET even with coarse statistical data, but better hardware support would surely help to allocate these cycles to the correct instructions.

8 Results

Testing the soundness of a WCET predictor is a tricky issue. To compare the computed value to a measured maximum execution time, we must force execution of the longest

path at runtime. Such control of execution may be difficult to achieve, especially for big applications, where it may be impossible to find the WCET by hand. By consequence the WCET that we can experimentally measure is an estimation and could be smaller than the real value.

We generated a series of short representative tests (with a unique execution path) to stress different characteristics of the processor and to check the approximations made by the predictor. This kind of tests provides a first feedback on the soundness of the predictor, and they are useful during the development of the tool, since the results and possible errors can be easily understood. These first tests include integer and floating-point matrix multiplications, the search for the maximum value in an array, solving differential equations with the Runge-Kutta method, polynomial evaluation with the Horner's rule, and distribution counting. Table 3 summarizes the results we obtained approximating the WCET of these tasks. We computed the predicted time using data gathered on the same environment used for the measured time tests, i.e. the standard XOberon system task set.

Table 3. Results for simple test cases.

Task	Measured time	Predicted time	Prediction error
Matrix multiplications	279.577 ms	310.533 ms	+11%
Matrix multiplications FP	333.145 ms	351.753 ms	+6%
Array maximum	520.094 ms	555.015 ms	+7%
FP array maximum	854.930 ms	814.516 ms	-5%
Runge-Kutta	439.905 ms	495.608 ms	+13%
Polynomial evaluation	1251.194 ms	1187.591 ms	-5%
Distribution counting	2388.816 ms	2579.051 ms	+8%

For the majority of the tests we achieved a prediction within 10% of the real value, and we avoid significant underestimations of the WCET.

The imprecision of some results is due to several factors. The biggest contributor is the inaccuracy of the runtime performance monitor, which clearly was not designed to meet our needs. The inability of the performance monitor to provide data on a more fine grained basis, e.g., such as basic blocks, forces us to use mean values. However, the combination of averages does not capture all aspects of the behavior of the complete program execution.

The importance of reducing the WCET by approximating some of the instruction length components with run-time information is shown in Table 4. The table shows the results obtained with conservative worst-case assumptions for the memory accesses: The obtained WCET is too high and unrealistic to be used in practice especially considering that many other aspects as the pipeline usage should be considered.

These benchmark tests do not say much about the usability of our predictor for real applications, because the code of these tests is quite homogeneous. Therefore, we profiled several real-time tasks of different real applications with a standard set of non–real-time processes handling user interaction and network communication.

Table 4. Approximations importance

Test	Matrix multiplications	Array maximum	Polynomial evaluation
WCET	298 ms	520 ms	1251 ms
Approximation	310 ms	555 ms	1188 ms
No cache hits	1403 ms	1901 ms	3193 ms

Testing big applications is clearly more difficult since the longest path is often unknown, and the maximum execution time must be forced by manipulating the program's input values.

Trigonometric functions are a good example of the problem: The predictor computes the longest possible execution, while, thanks to optimized algorithms, different lengths are possible at runtime. It is obviously difficult, or often even impossible, to compute an input set such that each call to these functions takes the maximum amount of time. We must therefore take into account that the experimentally measured maximum length is probably smaller than the theoretical maximum execution time. On the other hand, tests with big applications are good for demonstrating the usability of the tool in the real world.

In principle, these real-life programs could be tested using a "black box" approach with the execution time as result. Then testing consists of maximizing the result function, i.e., the dynamic path length, varying the input data and submission time [29,26]. Unfortunately, this approach is not feasible with real applications where the amount of input data is huge, and some execution paths may damage the hardware (robot) or operator. We therefore are required to exercise some discretion in controlling the execution of the application kernels. Tables 5 to 7 report the maximum measured execution time when trying to force the longest path and the predicted WCET approximation.

The first application tested is the control software of the *LaserPointer*, a laboratory machine that moves a laser pen applied on the tool-center point of a 2-joints (2 DOF) manipulator (Table 5).

Table 5. Results for *LaserPointer*.

Task	Measured WCET	Predicted WCET
Watchdog	290 μs	287 μs
Planner	297 μs	298 μs
Controller	730 μs	1071 μs

The second application being tested is the control software of the *Hexaglide*, a parallel manipulator with 6 DOF used as a high speed milling machine [11]. The two real-time tasks contain a good mix of conditional and computational code (Table 6).

The *Robojet* is a hydraulically actuated manipulator used in the construction of tunnels. This machine sprays liquid concrete on the walls of new tunnels using a jet as its tool. We profiled three periodic real-time tasks that compute the different trajectories and include complex trigonometric and linear algebra computations (Table 7).

Table 6. Results for *Hexaglide*.

Task	Measured WCET	Predicted WCET
Learn	65 μs	65 μs
Dynamics	286 μs	451 μs

Table 7. Results for *Robojet*.

Task	Measured WCET	Predicted WCET
TrajectoryQ	8 μs	8 μs
TrajectoryN	339 μs	399 μs
TrajectoryX	164 μs	183 μs

For the majority of the benchmark tests we get good results, with an imprecision in the order of 10 percent. The controller handler of the *LaserPointer* and the dynamics handler of the *Hexaglide* tests introduce some imprecision, due to the large number of trigonometric functions present in their code; different executions of these function have vastly varying execution times, and this property complicates the measurement of the real worst case.

In summary, for these real applications, our technique produces conservative approximations yet avoids noticeable underestimations of the WCET. These results demonstrate the soundness of the method for real-world products. If a user feels that the benchmark tests reflect the properties of future applications more than the three kernels presented here, then such a user can add a margin (of say 10%) to the approximated WCET.

Providing hints to compute the WCET for these tasks requires little effort. Only a few loops require hints to specify the bound of the trip count, and a simple recursive function was rewritten into a loop construct (see Table 8, which lists the number of source lines). For example, to profile the *Hexaglide* kernel, we only needed to specify two maximal loop iteration bounds and measure the length of four driver calls:

Table 8. Source code changes.

Application	Hints	Method length	Code size
LaserPointer	-	5	958 lines
Hexaglide	2	4	2226 lines
Robojet	17	-	1616 lines

9 Concluding Remarks

This paper describes an approach to compute a realistic value for the worst-case execution time for tasks that are executed on a system with preemptive multitasking and a dynamic constellation of running processes. Our approach combines compile-time analysis with

statistical performance data gathered by the hardware assists found in a state-of-the-art microprocessor. When the compiler is unable to automatically determine the trip count of a loop, or the path length of a method invocation, the user must provide a hint. These hints are used to determine the longest path of the task, but the compiler inserts code to monitor during execution in a real-time environment that the hints are accurate.

Our tool computes good approximations of the WCET of real-time processes of complex mechatronic applications in an unpredictable scheduling environment, avoiding the significant effort needed to obtain a similar value by trial-and-error experimentation or similar techniques. This solution is already employed in our robotics laboratory to compute the WCET of real products under development. Two features helped to gain user acceptance: this technique has minimal impact on the development time, and only minimal effort (occasional hints) is expected from the programmer.

Using a real processor (PowerPC 604e) for our experiments exposed us to shortcomings of its implementation: The hardware performance monitor provides only coarse-grain information, and the event model supported by these hardware components is not supportive of the execution of high-level language programs. Our tool must apply a number of approximations to map information gathered by the hardware monitors into values that can be used to compute the WCET. Although we succeed in producing a usable value, better hardware assists would simplify this part of our tool. If advanced high-performance consumer processors are to play a role in real-time systems, further improvements in the monitoring hardware assists are essential.

The principal benefit of the techniques described here is that they allow us to compute a good WCET approximation of a given program for a system that uses preemptive scheduling on an set of processes unknown at the analysis time and runs on a modern high-performance processor. Minimal user interaction makes this toolset suitable for application experts. This tool demonstrates that a few simple ideas, combined with a good model of the processor architecture, provide the basis of an effective system.

10 Acknowledgments

We would like to thank Richard Hüppi for supplying the sources of *LaserPointer* controller software, and Marcel Honegger for supplying the sources of the *Hexaglide* and *Robojet* software.

References

[1] P. Altenbernd. On the false path problem in hard real-time programs. In *Proc. 8th Euromicro Workshop on Real-Time Systems*, pages 102–107, L'Aquila, Italy, June 1996.

[2] R. Brega. A real-time operating system designed for predictability and run-time safety. In *Proc. 4th Int. Conf. Motion and Vibration Control (MOVIC)*, pages 379–384, Zurich, Switzerland, August 1998. ETH Zurich.

[3] J. Busquets-Mataix and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proc. 8th Euromicro Workshop on Real-Time Systems*, pages 271–276, L'Aquila, June 1996.

[4] C.-S. Cheng, J. Stankovic, and K. Ramamritham. Scheduling algorithms for hard real-time systems–A brief survey. In J. Stankovic and K. Ramamritham, editors, *Tutorial on Hard Real-Time Systems*, pages 150–173. IEEE Computer Society Press, 1988.

[5] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proc. 30th Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO-97)*, pages 292–302, Los Alamitos, CA, December 1997. IEEE Computer Society.

[6] D. Diez and S. Vestli. D'nia an object oriented real-time system. *Real-Time Magazine*, (3):51–54, March 1995.

[7] P. Emma. Understanding some simple processor-performance limits. *IBM J. Research and Development*, 43(3):215–231, 1997.

[8] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Trans. Computers*, 48(1):53–70, January 1999.

[9] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proc. 4th Real-Time Technology and Applications Symp.*, pages 12–21, Denver, Colorado, June 1998.

[10] C. Healy and D. Whalley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *Proc. 5th Real-Time Technology and Applications Symp.*, pages 79–88, Vancouver, Canada, June 1999.

[11] M. Honegger, R. Brega, and G. Schweitzer. Application of a nonlinear adaptive controller to a 6 dof parallel manipulator. In *Proc. IEEE Int. Conf. Robotics and Automation*, pages 1930–1935, San Francisco CA, April 2000. IEEE.

[12] IBM Microelectronic Division and Motorola Inc. *PowerPC 604/604e RISC Microprocessor User's Manual*, 1998.

[13] S.-K. Kim, S. Min, and Ha R. Efficient worst case timing analysis of data caching. In *Proc. 2nd 1996 IEEE Real-Time Technology and Applications Symposium*, pages 230–240, Boston, MA, June 1996. IEEE.

[14] D. Kirk. SMART (Strategic Memory Allocation for Real-Time) cache design. In *Proc. 10th IEEE Real-Time Systems Symp.*, pages 229–239, Santa Monica, California, December 1989. IEEE.

[15] E. Kligerman and A. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Trans. on Software Eng.*, 12(9):941–949, September 1986.

[16] L. Ko, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M. Harmon. Supporting the specification and analysis of timing constraints. In *Proc. 2nd IEEE Real-Time Technology and Applications Symp.*, pages 170–178, Boston, MA, June 1996. IEEE.

[17] C.-G. Lee, J. Hahn, Y.-M. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *Proc. 17th IEEE Real-Time Systems Symp.*, pages 264–274, Washington, D.C., December 1996. IEEE.

[18] F. Levine and C. Roth. A programmer's view of performance monitoring in the PowerPC microprocessor. *IBM J. Research and Development*, 41(3):345–356, May 1997.

[19] Y.-T. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond directed mapped instructions caches. In *Proc. 17th IEEE Real-Time Systems Symp.*, pages 254–263, Washington, D.C., December 1996. IEEE.

[20] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Proc. 3rd IEEE Real-Time Technology and Applications Symp.*, Montreal, Canada, June 1997. IEEE.

[21] S.-S. Lim, Y. Bae, G. Jang, B.-D. Rhee, S. Min, C. Park, H. Shin, K. Park, S.-M. Moon, and C.-S. Kim. An accurate worst case timing analysis for RISC processors. *IEEE Trans. on Software Eng.*, 21(7):593–604, July 1995.

[22] S.-S. Lim, J. Han, J. Kim, and S. Min. A worst case timing analysis technique for multiple-issue machines. In *Proc. 19th IEEE Real-Time Systems Symp.*, pages 334–345, Madrid, Spain, December 1998. IEEE.
[23] A. Mok and G. Liu. Efficient runtime monitoring of timing constraints. In *Proc. 3rd Real-Time Technology and Applications Symp.*, Montreal, Canada, June 1997.
[24] H. Mössenböck and N. Wirth. The programming language Oberon-2. *Structured Programming*, 12(4), 1991.
[25] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
[26] F. Müller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *Proc. 19th Real Time Technology and Applications Symp.*, pages 179–188, Madrid, Spain, June 1998. IEEE.
[27] C. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, (5):31–62, 1993.
[28] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *J. Real-Time Systems*, 1(2):160–176, September 1989.
[29] P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. In *Proc. 19th Real-Time Systems Symp.*, pages 134–143, Madrid, Spain, December 1998. IEEE.
[30] P. Puschner and A. Vrchoticky. Problems in static worst-case execution time analysis. In *9. ITG/GI-Fachtagung Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen, Kurzbeiträge und Toolbeschreibungen*, pages 18–25, Freiberg, Germany, September 1997.
[31] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *J. System Architecture*, 46(4):339–335, April 2000.
[32] H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proc. 19th IEEE Real-Time Systems Symp.*, pages 144–153, Madrid, Spain, December 1998. IEEE.
[33] N. Wirth and J. Gutknecht. *Project Oberon — The Design of an Operating System and Compiler*. ACM Press, New York, 1992.
[34] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, October 1993.

A Glossary

CPI Cycles per instruction: the total number of cycles needed to completely execute a given instruction, see (3).
$exec_{unit}$ The time, in cycles, needed by the execution unit to handle the instruction when no stalls occur, see (3).
FCE Finite cache effect: accounts, in cycles, for the effects of the memory hierarchy of the instruction length, see (1).
$idle_{unit}$ The average execution unit idle time per cycle, see Sect. 6.4.
$load_{unit}$ The number of instructions handled by the corresponding unit per cycle, see Sect. 6.4.
$miss_{load}$ The percentage of cache load misses.
$miss_{store}$ The percentage of cache store misses.
N_q The mean number of instructions waiting in the execution unit's reservation station, see (6).

p_{dep} The reported number of dependences per cycle in a given reservation station, see (7).

p_{event} The probability of a stall in the four-instruction dispatch queue, see (9).

$parallelism$ The number of instructions that are simultaneously in the corresponding execution unit, see (12).

$penalty_{load}$ The average number of additional cycles needed to fetch the data from the memory hierarchy, see (4).

$queue\_size$ The length of the dispatch queue, see (9)

$stall_{dispatch}$ The time, in cycles, a given instruction stalls in the dispatch unit, see (9).

$stall_{pipeline}$ The time, in cycles, a given instruction stalls outside the corresponding execution unit, see (9).

$stall_{unit}$ The time, in cycles, a given instruction stalls in the corresponding execution unit, see (5) and (8).

$util_{unit}$ The fraction of instructions handled by the corresponding unit, see (14).

ρ The unit utilization factor or arrival/service ratio, see (6).

A Design and Implementation of a Remote Debugging Environment for Embedded Internet Software

Kwangyong Lee, Chaedeok Lim, Kisok Kong, and Heung-Nam Kim

Embedded Software Research Team
Electronics and Telecommunications Research Institute
161 Kajong-Dong, Yusong-Gu, Taejon, 305-350, Korea
{kylee, cdlim, kskong, hnkim}@etri.re.kr

Abstract. It is necessary to use development tools in developing embedded real-time application software for Internet appliances. In this paper, we describe an integrated remote debugging environment for ***Q+(QPlus)*** real-time kernel which has been built for an embedded internet application. The remote development toolset called ***Q+Esto*** consists of several independent support tools: an *interactive shell*, a *remote debugger*, a *resource monitor*, a *target manager* and a *debug agent*. Using the ***remote debugger*** on the host, the developer can spawn and debug tasks on the target run-time system. It can also be attached to already-running tasks spawned from the application or from interactive shell. Applicatoin code can be viewed as C source, or as assembly-level code. It incorporates a variety of display windows for source, registers, local/global variables, stack frame, memory, event traces and so on. The ***target manager*** implements common functions that are shared by Esto tools, e.g., the host target communiction, object file loading, and management of target-resident host tool's memory pool and of target system symbol-table, and so on. These functions are called OPEN C APIs and they greatly improve the extensibility of the Esto Toolset. ***Debug agent*** is a daemon task on real-time operating systems in the target system. It gets requests from the host tools including debugger, interprets the requests, executes them and sends the results to the host.

1 Introduction

With the rapid growth of Internet, many devices such as Web TVs, PDAs, and Web phones, begin to be directly connected to the Internet. These devices need real-time operating systems (RTOS) to support complex real-time applications running on them. Development of such real-time applications called *embedded internet applications*, is difficult due to the lack of adequate tools, especially debuggers[1-3]. Many software development tools currently in the market are not effective for building real-time systems because the tools do not support real-time features[4,5]. While there are several real-time software development tools, which can provide these facilities, they cause heavy load on target systems with limited resources, e.g. memory, CPU performance, I/O, etc [6-8]. Therefore, suitable real-time software development tools which use only a small portions of target system's resources and provide platform-independence

J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 199-203, 2001.

are required to easily develop very complex real-time applications running on today's intelligent Internet devices [9,10].

In this paper, we introduce our approach to a debugging environment for a micro-computer embedded system such as internet applications. This debugging system is a remote debugging environment which includes an embedded system as a target system. The main features of our environment are as follows:

- easy-to-use user interfaces
- minimized access time of the target using target manager on a host
- make it simple to customize or extend the debugging facilities using modular & layered architecture
- effectively presents the content information so the user always knows where he is and how he got there

2 Q+Esto Remote Debugging Environment

Figure 1 is the whole figure of the remote debugging environment. We have implemented a prototype environment of the remote debugging environment in Windows NT, using Visual C++ and GNU gcc compiler. To communicate with the target system, we used an ONCRPC compatible RPC for Windows NT. The target operating system is Q+ running on Strong ARM 110 processor.

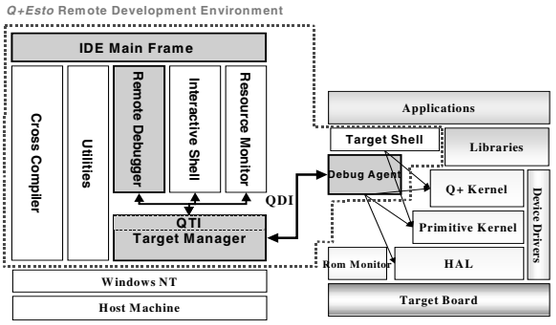


Fig. 1. Structure of Q+Esto

In the past, the structures of remote development environments are that each tool manages everything connecting to target system and accessing to target resources without any mediators between the tools and target system. In these structures, target system has very high communication overhead if the tools try to access the target system at the same time. And, the user of a tool is able to arise some severe faults in the target system, thus, the user of other tools cannot access to the target system until the target system is recovered from the fault. Currently, our approach is that has the middle components called the target manager and debug agent between the host tools and the target system. On the host, the target manager mediates the host-target communication, and manages only one communication channel from host to target, and manages the symbol table which is shared by the host tools, and manages the target-

resident host memory pool, and so on. We thought this structure is more efficient than the past one in that current one can be saving the target resource and the communication bandwidth relatively.

3 Designing and Implementing a Remote Debugging Environment

In the past, establishing communication between a debugger and an RTOS required a unique implementation for each debugger and each RTOS. Thus, sustantial redevelopment work was required for each new RTOS and each new debugger. To resolve this problem, our environment has adopted a common target interfaces by which the debugger software communicates to the RTOS.

Q+Esto Remote deubugger uses a graphical user interface with enhanced capabilities. We can open as many windows as we like and arrange them on our screen to suit our needs. Figure 2 shows the Output window, the Code window, the Memory window, Register window, Callstack window, the Variable window, and the Watch window.

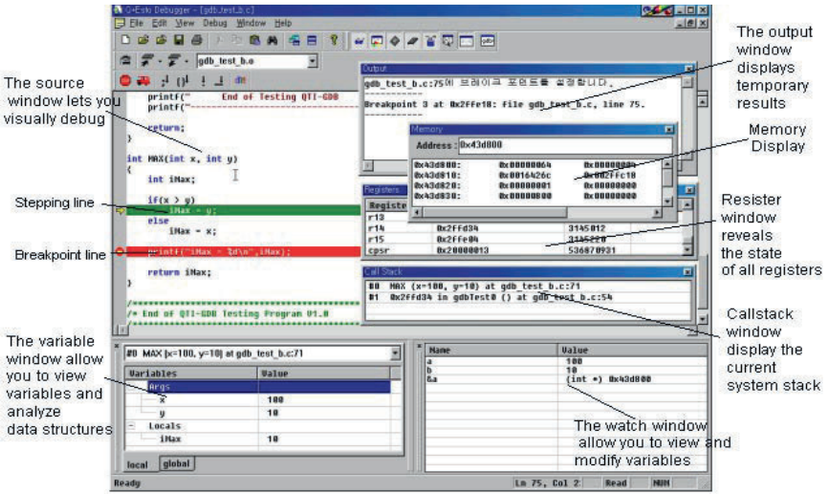


Fig. 2. Q+Esto Debugger GUIs

The Q+Esto **target manager** is the key component on the host system, and the foundation of the host-resident tools. There is one target manager per target; all host tools access the target through this component, which functions to satisfy the tool by breaking each request into the necessary transactions with the target agent. The target manager manages the details of the target connection-whichever method is chosen-so that individual tools need not concern themselves with host-to-target transport mechanisms. In some cases, target manager passes a tool's request directly to the target agent. In other case, request can be fulfilled entirely within the target manager on the host. The target manager also allocates target memory from a pool dedicated to the host tools, and manages the target's symbol table on the host. This permits the target

manager to do most of the work of dynamic linking(address resolution) on the host system, before downloading a new module to the target-resulting in a large reduction in total target requests. Q+Esto target manager has four subsystems, symbol table manager, object module manager, host-pool memory manager, and communication back-end manager. Figure 3 shows the target manager structure.

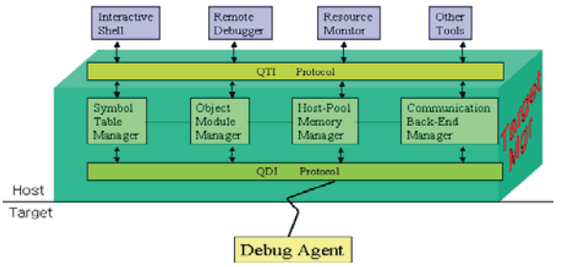


Fig. 3. Q+Esto Target Manager

As shown in figure 4, Q+Esto **debug agent** is a compact implementation of the core services necessary to respond to requests from the host tools. The agent responds to the requests transmitted by the target manager from the host tools. These requests include memory transactions, notification services for breakpoints and other target events, virtual I/O supports, and task control. The debug agent contains a lightweight implementation of UDP/IP, which supports an RPC-based messaging protocol, called *Q+ Debug Interface* (QDI). QDI protocol is developed by ETRI under the project of development of reconfigurable RTOS, called *Q+*. The protocol is designed to provide target-independence and uses minimum amount of resource in the target. It is a core minimum of the services necessary to respond to the requests from the host tools including debuggers. These requests include memory transactions, breakpoint setting, event notification services and task control. The QDI protocol uses External Data Representation (XDR) for the data transfer[12].

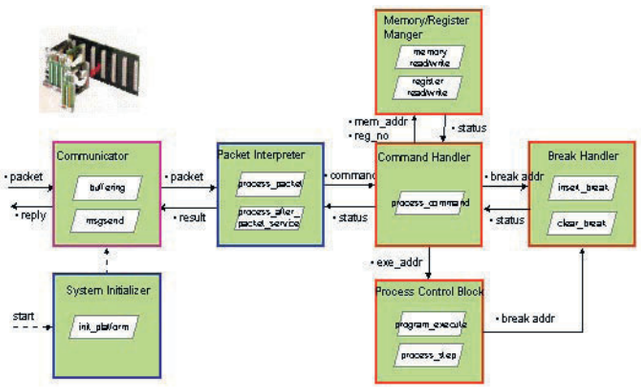


Fig. 4. Q+Esto Debug Agent

4 Conclusion and Future Works

In this paper, we describe real-time software development environment for developing embedded-systems such as Internet appliances. We propose the structure and functions of remote debugging environment supporting efficient cross development.

We are going to upgrade standard protocols to provide more efficiency to the development environment. Research on the reduction of the protocol overhead is an important issue. The amounts of messages between the host and the target have great impact on the performance of the development tools. In this respect, the numbers of functions in the QTI, and QDI protocols will be decreased.

Currently the remote debugger environment is implemented only for the StrongARM evaluation board [11]. We will port it to another CPU boards in order to enhance portability of the tools and to provide more generality to our development tools, called *Q+Esto*. Embedded software development tools that have hardware-independence and RTOS-independence will become more important in the era of post-PC.

References

1. Jack G. Ganssle, "Debuggers for Modern Embedded Systems," *Embedded Systems Programming*, Nov. 1998.
2. Jonathan B. Rosenberg, *How Debuggers Work*, John Wiley & Sons, 1996.
3. Hideyuki Tokuda and Makoto Kotera, "A Real-Time Tool Set for the ARTS Kernel," *Proceedings of Real-Time Systems Symposium*, 1988.
4. Eldad Maniv, "New Trends in Real-Time Software Debugging," Real-Time Magazine 99-2 (http://www.realtime-info.com), pp.23-25, 1999.
5. T. Yasuda, K. Ueki, "A Debugging Technique Using Event History", *Proc. of the Conference on Real-Time Computing Systems and Applications*, pp. 137-141, 1994.
6. WindRiver, *Tornado User's Guide,* 1995.
7. WindRiver, *Tornado API Guide 1.0.1*, 1997.
8. Microtec, *Spectra Boot and VRTX Real-Time OS*, 1996.
9. Eun-Hyang Lee, et al., "A Cross Debugging Architecture for Switching Software," *Proceedings of International Conference on Communication Technology* (ICCT), 1996.
10. YoungJoon Byun, et al., "High-Level CHILL Debugging System in Cross-Development Environments," *Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing*, 1998.
11. Intel, *StrongARM EBSA-285 Evaluation Board*, Oct. 1998.
12. Norishi Iga, et al., "Real-Time Software Development System RTIplus," *Proceeding of the 12th TRON Project International Symposium*, 1995.

Optimizing Code Size through Procedural Abstraction
(Extended Abstract)

Johan Runeson[1,2], Sven-Olof Nyström[1], and Jan Sjödin[1,2]

[1] Information Technology, Computing Science Department, Uppsala University,
Box 311, SE-751 05 Uppsala, Sweden,
{d96jru, svenolof, jans}@csd.uu.se
[2] IAR Systems,
Box 23051, SE-750 23 Uppsala, Sweden
{johanr,jans}@iar.se

Abstract. Memory size is an important economic factor in the development of embedded systems. It is therefore desirable to find compiler optimization techniques that reduce the size of the generated code. One such technique is *procedural* abstraction, where repeated occurrences of equivalent code fragments are factored out into new subroutines. Previously, procedural abstraction has been applied at the machine code level in optimizing linkers or binary rewriting tools. We investigate the prospects of applying procedural abstraction at the intermediate code level of a whole-program optimizing C compiler. Placing the optimization before register allocation makes it target indepentdent, and will allow us to take full advantage of powerful code selection, register allocation and code scheduling techniques.

1 Procedural Abstraction

A flow graph fragment can be replaced by a call to a new procedure if it has unique entry and exit edges, i.e., it is a single entry single exit region. The body of the new procedure should be a copy of the region, with all temporaries replaced by fresh temporaries. Parameter passing and (multiple) return values are used to transfer live values to and from the new temporaries.

Two regions can be replaced by calls to the same procedure if they are semantically equivalent. Since semantic equivalence is, in general, undecidable, we use a syntactic criterion, *matching*, which guarantees semantic equivalence.

2 Matching

Given two regions R and R' with nodes N and N', and with temporaries T and T', we say that the two regions *match* if there are bijective mappings $f : N \to N'$ and $g : T \to T'$ such that f maps the destination of the entry edge of R to the destination of the entry edge of R', and whenever $f(n) = n'$, all the following hold for the nodes n and n'

J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 204–205, 2001.

1. they have the same number of instructions,
2. the kth instruction of n is identical to the kth instruction of n' in everything except the operands, for $k \geq 0$, and
3. for each pair of corresponding operands, one of the following holds
   a) the operands are constants c and c' such that $c = c'$,
   b) they are temporaries t and t' such that $g(t) = t'$, or
   c) they are labels denoting nodes m and m' such that either m is the destination of the exit edge of R and m' is the destination of the exit edge of R', or $f(m) = m'$.

3 Algorithm

The test for matching can be performed during a depth-first traversal of the nodes of the two regions, building the mappings f and g incrementally as nodes and temporaries are encountered.

Since we may have to perform matching tests for all pairs of regions in the program, and the number of regions is potentially quadratic in the number of flow graph edges, it is essential that we find ways to reduce the number of necessary tests. This can be done in the following ways.

1. Consider only canonical single entry single exit regions. (The number of canonical regions is linear in the number of edges.)
2. Compute a fingerprint for each region, such that regions with different fingerprints are guaranteed not to match.
3. Use a UNION-FIND data structure to deduce matching results from already completed tests.
4. Use information about nesting relationships among regions to be able to re-use results from the matching of sub-regions when enclosing regions are matched.

4 Experiments

We have implemented a prototype of the procedural abstraction algorithm, and applied it to some standard benchmark programs. The intermediate code in the compiler is basically three-address code. Some global optimizations are performed by the frontend, but all code expanding transformations were disabled for this experiment.

The prototype allows us to measure the size of the program, in intermediate code instructions, before and after procedural abstraction. For the benchmarks used, the code size reduction ranges from 2.5 to 12.4 percent, with an average of 6.6 percent.

5 Further Information

A full version of this paper will be published as an ASTEC technical report, and made available at:
`http://www.docs.uu.se/astec/Reports/`.

Automatic Validation of Code-Improving Transformations*

Robert van Engelen, David Whalley, and Xin Yuan

Dept. of Computer Science, Florida State University, Tallahassee, FL 32306-4530
{engelen,whalley,xyuan}@cs.fsu.edu

Abstract. Programmers of embedded systems often develop software in assembly code due to critical speed and/or space constraints and inadequate support from compilers. Many embedded applications are being used as a component of an increasing number of critical systems. While achieving high performance for these systems is important, ensuring that these systems execute correctly is vital. One portion of this process is to ensure that code-improving transformations on a program will not change the program's semantic behavior. This paper describes a general approach for validation of many low-level code-improving transformations made either by a compiler or specified by hand.

1 Introduction

Software is being used as a component of an increasing number of critical systems. Ensuring that these systems execute correctly is vital. One portion of this process is to ensure that the compiler produces machine code that accurately represents the algorithms specified at the source code level. This is a formidable task since an optimizing compiler not only translates the source code to machine code, it may apply hundreds or thousands of compiler optimizations to even a relatively small program. However, it is crucial to try to get software correct for many systems. This problem is exacerbated for embedded systems development, where applications are often either developed in assembly code manually or compiler generated assembly is modified by hand to meet speed and/or space constraints. Code improving transformations accomplished manually are much more suspect than code generated automatically by a compiler.

There has been much work in the area of attempting to prove the correctness of compilers [3,4,6]. More success has been made in the area of validating compilations rather than the compiler itself [2]. Likewise, there has been progress in proving type, memory safeness, and other related properties of a compilation [7,8, 10]. Horwitz attempted to identify semantic differences between source programs in a simple high-level language containing a limited number of constructs [5]. In our approach we show the equivalence of the program representation before and after each improving transformation. While many code-improving transformations may be applied to a program representation, each individual transformation typically consists of only a few changes. Also, if there is an error, then the

* Supported by NSF grant CCR-9904943.

J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 206–210, 2001.

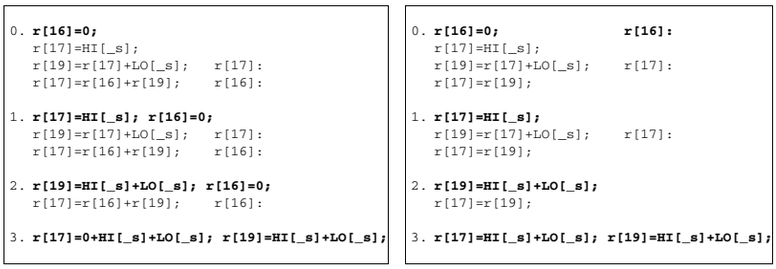


Fig. 1. Example Merging Effects within a Single Block

compiler writer or assembly programmer would find it desirable for a system to identify the transformation that introduced the error. For each code-improving transformation we only attempt to show the equivalence of the region of the program associated with the changes rather than showing the equivalence of the entire program representation. We show equivalence of the region before and after the transformation by demonstrating that the effects the region will have on the rest of the program will remain the same.

2 Implementation

We validate code-improving transformations in the *vpo* compiler [1], which uses RTLs (register transfer lists) to represent machine instructions. Each register transfer is an assignment that represents a single effect on a register or memory cell of the machine. Thus, the RTL representation served as a good starting point for calculating the semantic effects of a region. Merging the RTLs in a region obtains an order-independent representation of effects. Merging also eliminates the use of temporaries within the region. Fig. 1 displays an example of merging effects. Each RTL is merged into the effects one at a time. When the destination of an effect is no longer live, then the effect is deleted (the point where a register dies is depicted to the right of that RTL). For instance, step 2 in Fig. 1(a) deletes the effect that updates `r[17]` since the register is no longer live. The final effects of the old and new regions in Fig. 1 are identical after normalization.

We automatically detect changes associated with a transformation by making a copy of the program representation before each code-improving transformation and comparing the program representation after the transformation with the copy. Each region consists of a single entry point and one or more exit points. We find the closest block in the control-flow graph that dominates all of the modified blocks. This dominating block contains the entry point of the region. A separate set of effects is calculated for each exit point from the region. The old and new regions are considered equivalent only if for each exit point they have the same effects.

Loops can be processed innermost first. The effects of each node at the same loop level are calculated only after all of its non-back edge predecessors have been processed. Merging the effects across loop nesting levels requires calculating the effects of an entire loop. One issue that we address is the represenation of a recurrence, which involves the use of a variable or register that is set on a previous loop iteration. An induction variable is one example of a simple recurrence.

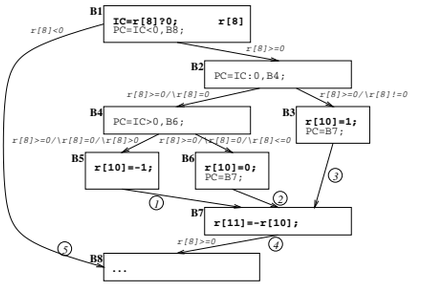
We modified the *vpo* compiler with calls to CTADEL to normalize effects. The CTADEL system [11] is an extensible rule-based symbolic manipulation program. The merging and subsequent normalization of effects results in canonical representations that enable a structural comparison to show that effects are semantically identical. The canonical representations of the effects corresponding to the exit points of old and new regions are compared by *vpo* to determine that the semantic effect of the transformed region of code is unchanged. The equivalence of the modified region is a sufficient condition for the correctness of a transformation, but it not a necessary condition.

The symbolic normalization of effects is illustrated by an example depicted in Fig. 2. The calculation of effects proceeds from the dominating block of the region (**B1**). The guarding conditions that result from conditional control flow are propagated down and used to guard RTLs. The final effects accurately describe the semantics of the region of code. Clearly, block **B5** is unreachable code. The *vpo* compiler applies dead-code elimination to remove block **B5**. The effect of the code after the transformation has been applied is unchanged (not shown). Hence, the dead-code elimination optimization is validated.

3 Results and Conclusions

A variety of types of transformations in the *vpo* compiler have been validated using our approach, including *algebraic simplification of expressions*, *basic block reordering*, *branch chaining*, *common subexpression elimination*, *constant folding*, *constant propagation*, *unreachable code elimination*, *dead store elimination*, *evaluation order determination*, *filling delay slots*, *induction variable removal*, *instruction selection*, *jump minimization*, *register allocation*, *strength reduction*, and *useless jump elimination*.

Table 1 shows some test programs that we have compiled while validating code-improving transformations. The third column indicates the number of improving transformations that were applied during the compilation of each program. The fourth column represents the percentage of transformations that we are able to validate. The only transformations that we cannot validate are those with regions that span basic blocks at different loop nesting levels since the ability to represent effects containing entire loops is in a development stage. The fifth column represents the average static number of instructions for each region associated with all code-improving transformations during the compilation. The final column denotes the ratio of compilation times when validating programs versus a normal compilation. The overhead is due to the application of sometimes tens of thousands of rewrite rules to normalize effects after a single transformation.



1. At (1) from block **B5** after merging: $\mathtt{r[10]} = \{-1\ if\ \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} > 0\}$
2. At (2) from block **B6** after merging: $\mathtt{r[10]} = \{0\ if\ \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} \leq 0\}$
3. At (3) from block **B3** after merging: $\mathtt{r[10]} = \{1\ if\ \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} \neq 0\}$
4. After combining the effects of the blocks **B5**, **B6**, and **B3** we obtain
$$\mathtt{r[10]} = \left\{ \begin{array}{ll} 1 & if\ \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} \neq 0 \\ 0 & if\ \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} \leq 0 \\ -1 & if\ \mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} > 0 \end{array} \right\} \stackrel{\text{simplify}}{=} \left\{ \begin{array}{l} 1\ if\ \mathtt{r[8]} > 0 \\ 0\ if\ \mathtt{r[8]} = 0 \end{array} \right\}$$
5. Merging the above effects with the effects of block **B7** yields
$$\mathtt{r[10]} = \left\{ \begin{array}{l} 1\ if\ \mathtt{r[8]} > 0 \\ 0\ if\ \mathtt{r[8]} = 0 \end{array} \right\};\ \mathtt{r[11]} = - \left\{ \begin{array}{l} 1\ if\ \mathtt{r[8]} > 0 \\ 0\ if\ \mathtt{r[8]} = 0 \end{array} \right\} \stackrel{\text{simplify}}{=} \left\{ \begin{array}{ll} -1 & if\ \mathtt{r[8]} > 0 \\ 0 & if\ \mathtt{r[8]} = 0 \end{array} \right\}$$
6. Merging the (empty) effects at transition (5) with the effects at transition (4) we obtain the effects of the region of code
$$\mathtt{r[10]} = \left\{ \begin{array}{ll} \left\{ \begin{array}{l} 1\ if\ \mathtt{r[8]} > 0 \\ 0\ if\ \mathtt{r[8]} = 0 \end{array} \right\} & if\ \mathtt{r[8]} \geq 0 \\ \mathtt{r[10]} & if\ \mathtt{r[8]} < 0 \end{array} \right\} \stackrel{\text{simplify}}{=} \left\{ \begin{array}{ll} 1 & if\ \mathtt{r[8]} > 0 \\ 0 & if\ \mathtt{r[8]} = 0 \\ \mathtt{r[10]} & if\ r[8] < 0 \end{array} \right\}$$
$$\mathtt{r[11]} = \left\{ \begin{array}{ll} \left\{ \begin{array}{ll} -1 & if\ \mathtt{r[8]} > 0 \\ 0 & if\ \mathtt{r[8]} = 0 \end{array} \right\} & if\ \mathtt{r[8]} \geq 0 \\ \mathtt{r[11]} & if\ \mathtt{r[8]} < 0 \end{array} \right\} \stackrel{\text{simplify}}{=} \left\{ \begin{array}{ll} -1 & if\ \mathtt{r[8]} > 0 \\ 0 & if\ \mathtt{r[8]} = 0 \\ \mathtt{r[11]} & if\ r[8] < 0 \end{array} \right\}$$

where the guard condition $\mathtt{r[8]} \geq 0$ is derived by forming the disjunction of the guard conditions on the incoming edges to block **B7**, which is the simplified form of
$(\mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} \neq 0) \vee (\mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} \leq 0) \vee (\mathtt{r[8]} \geq 0 \wedge \mathtt{r[8]} = 0 \wedge \mathtt{r[8]} > 0)$

Fig. 2. Example Normalization of Effects

However, this overhead would probably be acceptable, as compared to the cost of not detecting potential errors. Validation can also be selectively applied on a subset of transformations to reduce the overall compilation time.

To summarize our conclusions, we have demonstrated that it is feasible to use our approach to validate many conventional code-improving transformations. Unlike an approach that requires the compiler writer to provide invariants for each different type of code-improving transformation [9], our general approach was applied to all of these transformations without requiring any special information. We believe that our approach could be used to validate many hand-specified transformations on assembly code by programmers of embedded systems.

Table 1. Benchmarks

Program	Description	#Trans	#Validated Region	Overhead	
`ackerman`	Ackerman's numbers	89	100.0%	3.18	13.64
`arraymerge`	merge two sorted arrays	483	89.2%	4.23	63.89
`banner`	poster generator	385	90.6%	5.42	34.13
`bubblesort`	bubblesort on an array	342	85.4%	6.10	34.37
`cal`	calendar generator	790	91.1%	5.16	105.64
`head`	displays the first few lines of files	302	89.4%	8.42	152.64
`matmult`	multiplies 2 square matrices	312	89.7%	5.55	28.97
`puzzle`	benchmark that solves a puzzle	1928	78.5%	5.85	128.98
`queens`	eight queens problem	296	85.8%	6.79	73.65
`sieve`	finds prime numbers	217	80.6%	6.85	21.90
`sum`	checksum and block count of a file	235	91.9%	8.62	195.19
`uniq`	filter out repeated lines in a file	519	91.1%	4.21	163.26
Average		492	88.6%	5.87	84.64

References

[1] M. E. Benitez and J. W. Davidson. A Portable Global Optimizer and Linker. In *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pages 329–338, June 1988.

[2] A. Cimatti and et. al. A Provably Correct Embedded Verifier for the Certification of Safety Critical Software. In *International Conference on Computer Aided Verification*, pages 202–213, June 1997.

[3] P. Dybjer. Using Domain Algebras to Prove the Correctness of a Compiler. *Lecture Notes in Computer Science*, 182:329–338, 1986.

[4] J. Guttman, J. Ramsdell, and M. Wand. VLISP: a Verified Implementation of Scheme. *Lisp and Symbolic Computation*, 8:5–32, 1995.

[5] S. Horwitz. Identifying the Semantic and Textual Differences between Two Versions of a Program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, 1990.

[6] F. Morris. Advice on Structuring Compilers and Proving Them Correct. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 144–152, 1973.

[7] G. Necula. Proof-Carrying Code. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[8] G. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, 1998.

[9] M. Rinard and D. Marinov. Credible Compilation with Pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verfication*, 1999.

[10] D. Tarditi, J. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[11] R. van Engelen, L. Wolters, and G. Cats. Ctadel: A generator of multi-platform high performance codes for pde-based scientific applications. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 86–93, May 1996.

Towards Energy-Aware Iteration Space Tiling

M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and H.S. Kim

Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802-6106
WWW: http://www.cse.psu.edu/~mdl

Abstract. Iteration space (loop) tiling is a widely used loop-level compiler optimization that can improve performance of array-dominated codes. But, in current designs (in particular in embedded and mobile devices), low energy consumption is becoming as important as performance. Towards understanding the influence of tiling on system energy, in this paper, we investigate energy behavior of tiling.

1 Introduction

Energy consumption has become a critical design concern in recent years, driven mainly by the proliferation of battery-operated embedded devices. While it is true that careful hardware design [1] is very effective in reducing the energy consumed by a given computing system, it is agreed that software can also play a major role. In particular, the application code that runs on an embedded device is the primary factor that determines the dynamic switching activity, one of the key factors influencing dynamic power dissipation.

Given a large body of research in optimizing compilers that target enhancing the performance of a given piece of code (e.g., see [3] and the references therein), we believe that the first step in developing energy-aware optimization techniques is to understand the influence of widely used program transformations on energy consumption. Such an understanding would serve two main purposes. First, it will allow compiler designers to see whether current performance-oriented optimization techniques are sufficient for minimizing the energy-consumption, and if not, what additional optimizations are needed. Second, it will give hardware designers an idea about the influence of widely used compiler optimizations on energy-consumption, thereby enabling them to evaluate and compare different energy-efficient design alternatives with these optimizations.

In this paper, we focus our attention on iteration space (loop) tiling, a popular high-level (loop-oriented) transformation technique used mainly for optimizing data locality. This optimization is very important because it is very effective in improving data locality and it is used by many optimizing compilers from industry and academia. While behavior of tiling from performance perspective has been understood to a large extent and important parameters that affect its performance have been thoroughly studied and reported, its influence on system energy is yet to be fully understood. In particular, its influence on energy consumption of different system components (e.g., datapath, caches, main

J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 211–215, 2001.

memory system, etc.) has largely been ignored. Having identified loop tiling as an important optimization, in this paper, we evaluate it, with the help of our cycle-accurate simulator, SimplePower [4], from the energy point of view.

2 Platform

Our framework [4] includes a transition-sensitive, cycle-accurate datapath energy model that interfaces with analytical and transition-sensitive energy models for the memory and bus sub-systems, respectively. The datapath is based on the ISA of the integer subset of the SimpleScalar architecture and the modeling approach used in this tool has been validated to be accurate (average error rate of 8.98%) using actual current measurements of a commercial DSP architecture [2]. The memory system of SimplePower can be configured for different cache sizes, block sizes, associativities, write and replacement policies, number of cache sub-banks, and cache block buffers. SimplePower uses the on-chip cache energy model based on 0.8μ technology parameters and the off-chip main memory energy per access cost based on the Cypress SRAM CY7C1326-133 chip. The overall energy consumed by the system is calculated as the sum of the energies consumed on processor core, instruction and data caches, busses/interconnect, and main memory. In all the experiments, our default data cache is 4 KB, one-way associative (direct-mapped) with a line size of 32 bytes. The instruction cache that we simulated has the same configuration. All the reported energy values in this paper are in Joules (J).

3 Evaluation

3.1 Tiling Strategy

In these experiments, we measure the energy consumed by the matrix-multiply code (the `ijk` loop where `i` the outermost) tiled using different strategies. The last two graphs in Figure 1 show the total energy consumption of eight different versions of the matrix-multiply code (one original and seven tiled) for two different input sizes: N=50 and N=100. We observe that (for both the input sizes) tiling reduces the overall energy consumption of this code. The x-axis shows the indices of the loops tiled.

In order to further understand the energy behavior of these codes, we break down the energy consumption into different system components: datapath (processor core) and memory. As depicted in the first two graphs in Figure 1, tiling a larger number of loops in general increases the datapath energy consumption. The reason for this is that loop tiling converts the input code into a more complex code which involves complicated loop bounds, a larger number of nests, and macro/function calls (for computing loop upper bounds). All these cause more branch instructions in the resulting code and more comparison operations that, in turn, increase the switching activity (and energy consumption) in the

datapath. For example, when the input size is 50 (100), tiling only the `i` loop increases the datapath energy consumption by approximately 10% (7%).

When we consider the main memory energy, however, the picture totally changes (the results are not shown here due to lack of space). For instance, when we tile all three loops, the main memory energy becomes 35.6% (18.7%) of the energy consumed by the untiled code when the input size is 50 (100). This is due to reduced number of accesses to the main memory as a result of better data locality. In the overall energy consumption, the main memory energy dominates and the tiled versions result in significant energy savings.

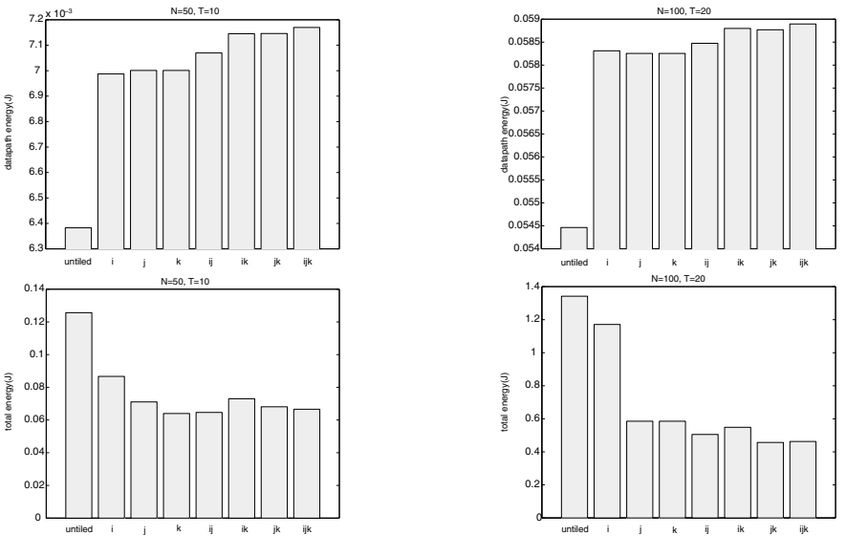


Fig. 1. Energy consumptions of different tiling strategies.

3.2 Tile Size Variations

We investigate the sensitivity of the energy behavior of tiling to the tile size. While the tile size sensitivity issue has largely been addressed in performance-oriented studies, the studies that look at the problem from energy perspective are a few. The results are given in Figure 2 for the input sizes 50 and 100. For each tiling strategy, we experiment with five different tile sizes (for N=50, the tile sizes are 2, 5, 10, 15, and 25, and for N=100, the tile sizes are 5, 10, 20, 25, and 50, both from left to right in the graphs for a given tiling strategy). We make several observations from these figures. First, increasing the tile size reduces the datapath energy and instruction cache energy as smaller the number

of tiles, the less complex the dynamic code is. However, as in the previous set of experiments, the overall energy behavior is largely determined by the energy spent in main memory. Since the number of accesses to the data cache is almost the same for all tile sizes (in a given tiling strategy), there is little change in data cache energy as we vary the tile size. It is also important to see that the energy performance of a given tiling strategy depends to a large extent on the tile size. For instance, when N is 50 and both `j` and `k` loops are tiled, the energy consumption can be as small as 0.064J or as large as 0.128J depending on the tile size chosen. It can be observed that, for each version of tiled code, there is a most suitable tile size beyond which the energy consumption starts to increase. Moreover, the most suitable tile size (from energy point of view) depends on the tiling style used. For instance, when using the `ij` version (with N=50), a tile size of 10 generated the best energy results, whereas with the `ik` version the most energy efficient tile size was 5.

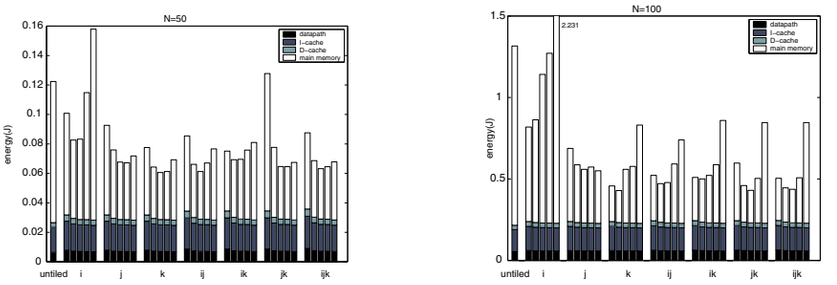


Fig. 2. Energy sensitivity of tiling to the tile size (blocking factor).

3.3 Input Size Variations

We now vary the input size (N parameter) and observe the variance in energy consumption of the untiled code and a specific tiled code in which all three loops in the nest are tiled (i.e., the `ijk` version). Specifically, we would like to observe the benefits (if any) of tailoring the tile size to the input size. The input sizes used are 50, 100, 200, 300, and 400. For each input size, we experimented with five different tile sizes (5, 10, 15, 20, and 25). The results, given in Table 1, show that for each input size there exists a best tile size from energy point of view. Further, this best tile size is different for each input size. To be specific, the best possible tile sizes (among the ones we experimented) for input sizes of 50, 100, 200, 300, and 400 are 10, 20, 20, 15, and 25, respectively. Consequently, it may not be a very good idea from energy point of view to fix the tile size at specific values.

Table 1. Energy consumption of tiling with different input sizes. `dp`, `ic`, `dc`, and `mem` denote the energies spent in datapath, instruction cache, data cache, and main memory, respectively.

N		tile size					
		untiled	5	10	15	20	25
50	dp	0.006	0.008	0.007	0.007	0.007	0.007
	ic	0.017	0.019	0.018	0.018	0.018	0.018
	dc	0.003	0.004	0.004	0.004	0.004	0.004
	mem	0.096	0.038	0.034	0.036	0.035	0.039
100	dp	0.054	0.063	0.060	0.060	0.059	0.059
	ic	0.135	0.151	0.146	0.145	0.144	0.143
	dc	0.026	0.031	0.029	0.029	0.028	0.028
	mem	1.100	0.259	0.210	0.217	0.206	0.277
200	dp	0.385	0.484	0.467	0.464	0.459	0.460
	ic	1.078	1.209	1.166	1.156	1.148	1.145
	dc	0.213	0.248	0.233	0.229	0.226	0.225
	mem	11.003	3.080	2.635	2.683	2.586	3.287
300	dp	1.388	1.704	1.648	1.642	1.627	1.632
	ic	3.638	4.080	3.937	3.895	3.875	3.864
	dc	0.722	0.836	0.785	0.770	0.763	0.760
	mem	42.859	9.448;	8.033	7.618	7.951	9.365
400	dp	3.166	3.962	3.849	3.826	3.791	3.795
	ic	8.621	9.671	9.331	9.238	9.185	9.158
	dc	1.707	1.982	1.859	1.826	1.808	1.799
	mem	96.087	22.324	18.385	18.666	17.984	15.041

4 Conclusions

In this paper, we study the energy behavior of tiling considering both the entire system and individual components such as datapath and memory. Our results show that the energy performance of tiling is very sensitive to input size and tile size. In particular, selecting a suitable tile size for a given computation might involve tradeoffs between energy and performance. We find that tailoring tile size to the input size generally results in lower energy consumption than working with a fixed tile size. This paper presents a preliminary study of energy behavior of tiling and further work is needed in this area.

References

1. J. Bunda, W. C. Athas, and D. Fussell. Evaluating power implication of CMOS microprocessor design decisions. In Proc. the 1994 International Workshop on Low Power Design, April 1994.
2. R. Y. Chen, R. M. Owens, and M. J. Irwin. Validation of an architectural level power analysis technique. In Proc. the 35th Design Automation Conference, June 1998.
3. M. Wolfe. High Performance Compilers for Parallel Computing, Addison Wesley, CA, 1996.
4. W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of SimplePower: a cycle-accurate energy estimation tool. In Proc. the 37th Design Automation Conference, Los Angeles, CA, June 5–9, 2000.

An Integrated Push/Pull Buffer Management Method in Multimedia Communication Environments

Sungyoung Lee, Hyon Woo Seung, Tae Woong Jeon

Department of Computer Engineering, KyungHee University, Seoul, Korea
{sylee@oslab.kyunghee.ac.kr}
Department of Computer Science, Seoul Women's University, Seoul, Korea
{hwseung@swu.ac.kr}
Department of Computer Science, Korea University, Seoul, Korea
{jeon@kusccgx.korea.ac.kr}

Abstract. Multimedia communication systems require not only high-performance computer hardware and high- speed networks, but also a buffer management mechanism to process voluminous data efficiently. Two buffer handling methods, push and pull, are commonly used. In the push method, a server controls the flow of data to a client, while in the pull method, a client controls the flow of data from a server. These two buffering schemes can be applied to the data transfer between the packet receiving buffer, which receives media data from a network server, and media playback devices, which play the received media data. However, the buffer management mechanisms at client-side mainly support only one of the push and the pull methods. In other words, different types of playback devices separately use either but not both of the buffer methods. This leads to inefficient buffer memory usage and to inflexible buffer management for the various types of media playback devices. To resolve these problems, in this paper we propose an integrated push/pull buffer mechanism able to manage both push and pull schemes in a single buffer at client-side. The proposed scheme can support various media playback devices using a single buffer space, which in consequence saves memory space compared to the case where a client keeps two types of buffers. Moreover, it facilitates the single buffer as a mechanism for absorbing network jitter effectively and efficiently. The proposed scheme has been implemented in an existing multimedia communication system called ISSA developed by the authors, and has showed good performance compared to the conventional buffering methods in multimedia communication environments.

1 Introduction

A flexible buffer management mechanism is required to process buffer I/O's efficiently, to calculate an adequate buffer size according to the application service and the media type, and to control the buffer over/underflow effectively.

Two buffer handling methods, Server-push (or just push) and Client-pull (or just pull), are commonly used, depending on which side has control over the data flow. In the push method, a server controls the flow of data and periodically transfers appropriate data to clients. Although the push method is suitable for broadcast services, the server must control the data transmission speed (bit-rate) in order to avoid buffer over/underflow at the clients' side. The pull method is a kind of polling method where a client, having control over the data flow, requests data to a server and the server transfers the requested data to the client. Although the client can control buffer over/underflow, the pull method is suitable only for unicast services, not for broadcast services [1,2,3].

The two methods can be applied to data transfer not only between network server and clients, but also between the packet receiving buffer, which receives media data from the network server, and media playback devices, which play the received media data. However, most buffer management mechanisms at the clients' side are usually focused on network considerations such as the buffer size control according to end-to-end network situations, and mainly support only one of the push and the pull methods. Consequently, they have some limitations in supporting various media playback devices. Even though some of them support both methods, it is difficult to utilize a variety of devices since they do not provide a unified structure. In an integrated multimedia communication system such as ISSA (Integrated Streaming Services Architecture)[4], various media playback devices cannot be supported if the packet receiving buffer provide either of the push or pull modes. A flexible buffering mechanism is also needed to absorb end-to-end network jitter, in a network environment like Internet with frequent packet loss and unstable bandwidth.

In this paper, we propose an efficient and flexible push/pull buffer management mechanism for the client-side, which readily supports various media playback devices by providing a unified interface for both push and pull modes at the client's packet receiving buffer, and easily absorbs end-to-end network jitter. Using the proposed scheme, can obviate the need to install an extra stub for buffer passing appropriate to each media device; furthermore, better efficiency in memory use can be accomplished by also letting the buffering function absorb network jitter, instead of buffering only.

This paper is organized as follows. We sketch the design architecture of the proposed buffer management scheme in Section 2. The implementation and experimental results of the scheme are described in Section 3. After briefly reviewing related work in Section 4, we present our conclusions in Section 5.

---

* This work was partly supported by Grant IJRP-9803-6 from the Ministry of Information and Communication of the Republic of Korea

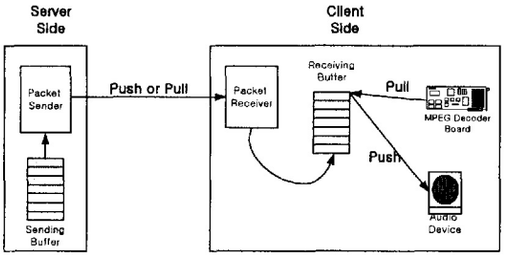
J. Davidson and S.L. Min (Eds.): LCTES 2000, LNCS 1985, pp. 216–220, 2001.

2 Design of the Proposed Scheme

In this section, we explain the operational architecture of the proposed scheme and the structure of the internal buffers, and summarize the buffer management algorithms.

2.1 The Design Model

Figure 1 illustrates the operational architecture of the push/pull buffer management scheme. It shows how the client-side push/pull scheme works with various kinds of media playback devices. The packets transmitted by the Packet Sender from the server through push or pull method are received and translated by the Packet Receiver of the client. Then, the media stream data are sent to the Receiving Buffer, and reproduced by the media playback devices through either the push or pull method initially set. To each of the media playback devices, an appropriate data transmission method is initially assigned. When the transmission session begins, the data stream is transmitted to the corresponding device, according to the method previously set. The Receiving Buffer on the client-side works regardless of the network media transmission methods, since the Packet Receiver receives data from the server through push or pull mode and sends the data to the buffer. The proposed scheme supports both push and pull mode in a single buffer.



[Figure 1] Operational Architecture of Proposed Scheme

2.2 Algorithms

The proposed buffer management scheme consists of 7 algorithms; two for buffer initialization and removal, another two buffering algorithms to absorb network jitter, and three algorithms for the data I/O.

Figure 2 shows the pseudo-code for the *InitBuffer* algorithm. It must first be determined which buffer passing mode (push or pull) the media device supports. Then, an appropriate buffer size is calculated using the following formula: *buffer size = bitrate of given media * buffering time * scale factor* where *bitrate of given media* is the playback speed of the given media (bit/secs), *buffering time* is the buffering time in seconds for the media, and *scale factor* is used to calculate an actual buffer size. The scale factor is used to get a "safe" buffer size to prevent overflow of the receiving buffer, and is usually greater than 1.0. If the factor is 1.0, the actual buffer size is equal to the ideal buffer size. After the buffer size is calculated, memory allocation is executed for the buffer. Then, the *StartBuffering* algorithm is performed.

The *DeinitBuffer* algorithm, shown in Figure 3, is used to stop media playback and remove the buffer. If buffering is in process (buffering mode is *start*), the *StopBuffering* algorithm is invoked. Then, waiting until all media stream data in buffer are consumed, it frees memory for the buffer.

```
InitBuffer Algorithm
Determine buffer passing mode for media device to use
    *) buffer passing mode = push or pull
Calculate buffer size
    *) buffer size = bitrate of the given media *
                     buffering time * scale factor
Allocate memory for buffer
Invoke StartBuffering algorithm
```

[Figure 2] InitBuffer Algorithm

```
DeinitBuffer Algorithm
If buffering mode == start then
        Invoke StopBuffering algorithm
Wait until all media stream data in buffer are
consumed
Free memory for buffer
```

[Figure 3] DeinitBuffer Algorithm

The *StartBuffering* algorithm begins by changing the buffering mode to *start*. Then, the current time is saved to calculate the playback delay time. The push/pull operations are locked until the total size of data in the buffer exceeds the buffering size.

```
StartBuffering Algorithm
Change buffering mode to start
Save current time for calculation of playback delay time
Lock push/pull operations until buffer data size exceed
buffering size
    *) buffer data size = total size of data in buffer
    *) buffering size = bitrate of the given media*
        buffering time
Invoke StopBuffering algorithm
```

[Figure 4] StartBuffering Algorithm

```
Stop Buffering Algorithm
Update playback delay time by adding additional
    playback delay time
    *) additional playback delay time =
        current time-saved buffering start time
    *) playback delay time = playback delay time +
        additional playback delay time
Unlock push/pull operations
Change buffering mode to stop
```

[Figure 5] StopBuffering Algorithm

Then, *StopBuffering* is invoked[Figure 4]. The buffering size is calculated using the following formula: *buffering size = bitrate of the given media * buffering time*

The *StopBuffering* algorithm, shown in Figure 5, is invoked at the end of *StartBuffering*. First, it calculates the additional playback delay time by subtracting the saved buffering start time from the current time, and updates the playback delay time by adding the additional playback delay time. Next, it unlocks push/pull operations by setting up a timer event for the next push mode operation if the current buffer passing mode is *push*, or by waking up the sleeping pull mode operation if the mode is *pull*. Finally, it changes the buffering mode to *stop*.

The *AddData* algorithm is shown in Figure 6. First, it checks whether buffer overflow has occurred. If so, it drops new incoming data and terminates the execution. If not, it copies new incoming data into the buffer. Then, if the buffer passing mode is *push*, it appends the info block corresponding to the new incoming data to the info block linked list.

Note: In terms of the QoS (Quality of Service), it is not desirable for the algorithm to drop any incoming data in case buffer overflow occurs. To resolve this problem, we could introduce of a scheme that saves the surplus data in a temporary buffer and moves them to the real buffer later. However, due to the overhead to manage the temporary buffer, real-time processing of the media data may not be guaranteed.

Figure 7 shows the *pushData* algorithm which sends the data in buffer to the media playback device in push mode. First, it checks whether buffer underflow has occurred. If so, it restarts buffering and terminates the execution. If not, it pushes the data to the media device and removes the data from the buffer by changing the pointer to stream head. Next, it sets up a timer event for the next *pushData* operation.

```
AddData Algorithm
Lock buffer
Apply playback delay time to calculate expire time of
    new incoming data
If (size of new incoming data + buffer data size >
        buffer size) then
        Buffer overflow occurred
        Drop new incoming data
        Unlock buffer
        Terminate this algorithm
Copy new incoming data into Buffer
Adjust pointer to stream tail with position of new data
If (buffer passing mode == push) then Add an info
    block corresponding to new incoming data.
Update buffered data size
Unlock buffer
```

[Figure 6] AddData Algorithm

```
pushData Algorithm
Lock buffer
If (buffer data size < size of data to push) then
        Buffer underflow occurred
        Invoke StartBuffering algorithm
        Terminate this algorithm
Push data of stream head in buffer to media device
Remove info block corresponding to removed data
Change pointer to stream head in buffer
Set up next timer event for pushData algorithm
Unlock buffer
```

[Figure 7] pushData Algorithm

Figure 8 shows the *pullData* algorithm. First, it checks whether buffer underflow has occurred. If so, it restarts buffering and waits until buffering stops. If not, it pulls the data to the media device and removes the data from the buffer by changing the pointer. Unlike *pushData*, *pullData* is invoked from the media device interface without using a timer event.

```
PullData Algorithm
Lock buffer
If (buffer data size < size of data to pull) then
Buffer underflow occurred
Invoke StartBuffering algorithm
Sleep until buffering stop
Pull data of stream head in buffer to media device
Change pointer to stream head in buffer
Unlock buffer
```

[Figure 8] pullData Algorithm

3 Performance Evaluation

In this section, we explain the implementation process and experimental results of the proposed scheme.

3.1 Implementation Environment

Our scheme has been implemented in an existing multimedia communication system, called ISSA [4] in which various streaming applications such as VOD and AOD can be easily produced. ISSA provides many functions concerned with content management, transmission protocols and media processing for various streaming application programs. Among many components in the ISSA, *Media Manager* plays an important role offering various media processing functions such as media file processing (MPEG, WAV, AU, etc.), database-stored media stream processing, A/V Codec for encoding/decoding media data to other formats, and controlling A/V devices. It consists of *Media Source* and *Media Sink*.

Since the *Media Sink* component plays the role of receiving the media stream data from the client, and sending them to the appropriate media playback devices, the proposed buffer management scheme is designed and implemented to be incorporated in *Media Sink*. In order to test the performance of the scheme, we built a simple server program sending media stream data stored in files, and a client program playing the data. Both programs are written in C++. The server program works on MS Windows-NT or Sun Solaris 2.X, and the client program on MS Windows-NT. The data transmission between the server and the client is done using the RTP (Real-Time Transport Protocol)[9] which was made for the real-time media data transmission over a TCP/IP-based 10/100 Mbps Ethernet.

3.2 Experimental Results

In order to evaluate the performance and measure the overhead of the proposed scheme implemented in ISSA, we have performed three experiments. In the first two experiments, real media playback devices were used to test whether the data transmission and buffering in the system works well with both push and pull methods. Tables 1 and 2 show the attributes of the media data used in the experiment. The network testbed used to test the remote playback function when the server and the client are on separate hosts.

[Table 1] Attributes of Test media A used in Experiments

Encoding Type	MPEG-1 System Layer Stream
Bitrate	1,715,200 bps
Resolution	320x240

[Table 2] Attributes of Test media B used in Experiments

Encoding Type	Linear PCM Audio
Bitrate	1,411,200 bps
Sampling Rate	44,100 Hz
Number of Channels	2
Bits Per Sample	16

Experiment 1

In the first experiment performed upon the testbed, we tested whether the proposed buffer management scheme operates correctly in pull mode. The scheme in the client-side sends the test media data in the receiving buffer, with the attributes in Table 2, to the DirectShow media playback device in pull mode. Table 3 shows the parameter values for Experiment 1. For each of 3 and 5 second Buffering Time (BT), we examined the operation of the scheme in pull mode with two cases of the Buffer Scale Factor(BSF), 1.1 and 1.3.

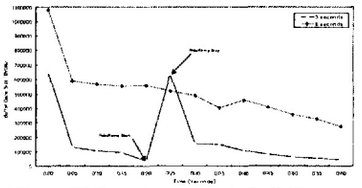
[Table 3] Buffer Parameter Values for Experiment 1

BT	3 seconds		5 seconds	
Buffering Size	1,715,200 bits / 8*3 = 643,200 bytes		1,715,200 bits / 8*5 = 1,072,000 bytes	
BSF	1.1	1.3	1.1	1.3
Buffer Size	643,200* 1.1 = 707,520 bytes	643,200* 1.3 = 836,160 bytes	1,072,000* 1.1 = 1,393,600 bytes	1,072,000* 1.3 = 1,393,600 bytes

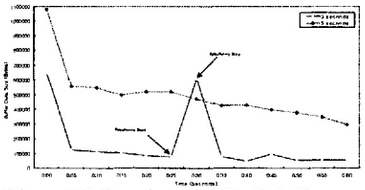
[Table 4] Buffer Parameter Values for Experiment 2

BT	3 seconds		5 seconds	
Buffering Size	1,411,200 bits/8*3 = 529,200 bytes		1,411,200 bits/8*5 = 882,000 bytes	
BSF	1.1	1.3	1.1	1.3
Buffer Size	529,200* 1.1 = 582,120 bytes	529,200* 1.3 = 687,960 bytes	882,000* 1.1 = 970,200 bytes	882,000 * 1.3 = 1,146,600 bytes

As shown in Figures 9 and 10, rebuffering caused by network packet delay occurred once during 1 minute of playing time in the case of 3-second Buffering Time, while no rebuffering occurred and the Buffer Data Size continuously decreased in the case of 5-second Buffering Time. The transmission delay time can be very long for high-bandwidth media streams like the MPEG-1 System Layer stream.



[Figure 9] Experiment 1 (A): Pull, Remote Playback, Buffer Scale Factor (1.1)



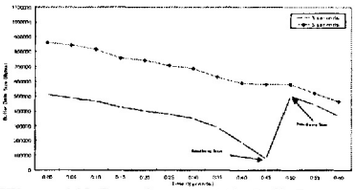
[Figure 10] Experiment 1 (B): Pull, Remote Playback, Buffer Scale Factor (1.3)

On the other hand, changing Buffer Scale Factors had little influence on the operation of the scheme. The reason was that the media processing speed in the client side was so fast that the data in the buffer could have been processed in time without delay

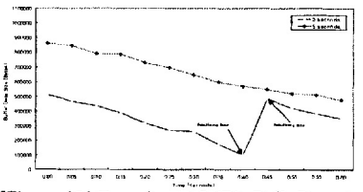
Experiment 2

In the second experiment that was performed upon the testbed as in the first experiment, we tested whether the proposed buffer management scheme operates correctly in push mode. The scheme in the client-side sends the test media data in the receiving buffer, with the attributes in Table 2, to the PCM audio playback device in push mode.

Table 4 shows the parameter values for Experiment 2. As in Experiment 1, for each of 3 and 5 second Buffering Time, we examined the operation of the scheme in push mode with two cases of the Buffer Scale Factor, 1.1 and 1.3. As can be seen in Figures 11 and 12, rebuffering occurred once in the case of 3-second Buffering Time, while no rebuffering occurred in the case of 5-second Buffering Time. From the first and second experiments, it can be said that the proposed scheme can absorb the network jitter regardless of the data passing mode of the buffer (push or pull), providing flexible buffering functions to overcome buffer over/underflow.



[Figure 11] Experiment 2 (A): Pull, Remote Playback, Buffer Scale Factor (1.1)



[Figure 12] Experiment 2 (B): Pull, Remote Playback, Buffer Scale Factor (1.3)

4 Related Work

Although much work has been done in the area of client-side buffer management techniques, not much research has been reported on push/pull mode buffer management related to data passing to media playback devices.

The JMF (Java Media Framework) of Sun Microsystems[5] offers push/pull buffer management methods through two interfaces, pushDataSource and pullDataSource. It is, however, not easy to program multimedia applications using JMF since it is not furnished with a unified interface for push/pull methods.

Acharya et al. at Brown University have studied ways to efficiently combine the push and pull approaches[6]. This method is focused upon supporting both push and pull methods among network server and clients, where the pull method is provided simply to make up for the weak points of the push method. In that sense, it is different from the scheme proposed in this paper which provides a unified structure to support both methods on the client-side.

The buffer management scheme proposed in DAVIC (Digital Audio VIsual Council)[9] is quite similar to the one proposed at Brown University[6]. However, the clients in the scheme not only request data through the back-channel, but also notify the server of the buffer states and gather the data needed in advance.

Jack Lee at Hong Kong University has proposed an efficient buffer management scheme in which the optimal buffer size is calculated to prevent server's buffer underflow and clients' buffer overflow, in his study of a concurrent push scheduling algorithm for push-based parallel video servers[7]. Even though this scheme shows a good performance by utilizing a parallel transmission method, it requires higher buffer capacity. Again, these two methods are different from the one we propose.

5 Conclusion

In this paper, we have proposed an efficient and flexible push/pull buffer management mechanism for the client-side, which readily supports various media playback devices by providing a unified interface for both push and pull mode at the client's packet receiving buffer, and easily absorbs end-to-end network jitter. Moreover, the proposed scheme can save memory space compared to the case where a client keeps two types of buffers. We have implemented the scheme on the ISSA system and performed six experiments whose results showed good performance with little overhead.

We are planning to expand the proposed scheme to a buffer management scheme which works among a network server and clients, not just between the client's packet receiving buffer and media playback devices. Also, more study is needed to handle buffer overflow more efficiently without large overhead.

References

[1] M. Franklin, and S. Zdonik, "Data In Your Face: push Technology in Perspective", *In Proc. of the ACM SIGMOD International Conference on the Management of Data*, Vol. 27, No. 2, pp. 516-521, June, 1998.

[2] S. Rao, H. Vin, and A. Tarafdar, "Comparative Evaluation of Server-push and Client-push Architectures for Multimedia Servers", *In Proc. of the 6th International Workshop on Network and Operating System Support for Digital Audio and Video*, April, 1996.

[3] J. P. Martin-Flatin, "push vs. pull in Web-Based Network Management", *In Proc. of the Integrated Network Management VI*, pp. 3-18, May, 1999.

[4] C.G. Jeong, H.I. Kim, Y.R. Hong, E.J. Lim, S Lee, J. Lee, B.S Jeong, D.Y. Suh, K.D. Kang, John A. Stankovic, Sang H. Son, "Design for an Integrated Streaming Framework", *Department of Computer Science, University of Virginia Technical Report*, CS-99-30, November, 1999.

[5] Sun Microsystems, Java Media Framework API Guide, September, 1999, http://java.sun.com/products/java-media/jmf/index.html.

[6] S. Acharya, M. Franklin, and S. Zdonik, "Balancing push and pull for Data Broadcast", *In Proc. of ACM SIGMOD Conference*, Vol. 26, No. 2, pp. 183-198, May, 1997.

[7] Jack Y. B. Lee, "Concurrent push-A Scheduling Algorithm for push-Based Parallel Video Servers", *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 9, No. 3, pp. 467-477, April, 1999.

[8] H. Schulzrinne, S. Casner, R. Frederick, and V. /Jacobson, RTP: A Transport Protocol for Real-Time Applications, *IETF RFC 1889*, January 1996.

[9] Delivery System Architecture And Interfaces, *ftp://ftp.davic.org/Davic/Pub/ Spec1_2/part04.pdf*

Author Index