Sixth Doctoral Workshop on Mathematical and Engineering Methods in Computer Science
(Selected Papers)

MEMICS'10, October 22–24, 2010, Mikulov, Czech Republic

Edited by
Luděk Matyska
Michal Kozubek
Tomáš Vojnar
Pavel Zemčík
David Antoš

OASIcs – Vol. 16 – MEMICS'10 www.dagstuhl.de/oasics

Editors

Luděk Matyska, Michal Kozubek, David Antoš
Masaryk University
Brno, Czech Republic
ludek@ics.muni.cz
kozubek@fi.muni.cz
antos@ics.muni.cz

Tomáš Vojnar, Pavel Zemčík
Brno University of Technology
Brno, Czech Republic
vojnar@fit.vutbr.cz
zemcik@fit.vutbr.cz

ACM Classification 1998
B.1 Control structures and microprogramming, B.4 Input/output and data communications, B.7 Integrated circuits, B.8 Performance and reliability, C.3 Special-purpose and application-based systems, D.1 Programming techniques, D.2 Software engineering, D.3 Programming languages, D.4 Operating systems, F.3 Logics and meanings of programs, F.4 Mathematical logic and formal languages, G.2 Discrete mathematics, H.5 Information interfaces and presentation, I.1 Symbolic and algebraic manipulation, I.2 Artificial intelligence, I.3 Computer graphics, I.4 Image processing and computer vision, I.6 Simulation and modeling, K.6 Management of computing and information systems

ISBN 978-3-939897-22-4

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik gGmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany.

Publication date
March, 2011.

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/OASIcs.MEMICS.2010.i

ISBN 978-3-939897-22-4 **ISSN 2190-6807** **www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 2190-6807

www.dagstuhl.de/oasics

Contents

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

In 2010, the International Doctoral Workshop on *Mathematical and Engineering Methods in Computer Science* (MEMICS'10) entered in the second pentad of three day workshops organized in the Czech Republic region of South Moravia by Faculty of Information Technology of Brno University of Technology and Faculty of Informatics of Masaryk University. MEMICS'10 returned to Mikulov, Czech Republic, and was held between October 22nd and 24th, 2010.

The MEMICS workshops provides an opportunity for PhD students to present and discuss their work in an international environment. The focus on PhD studies and not a particular narrow scientific area leads to a cross-disciplinary orientation of MEMICS workshops, providing a pleasant environment for an exchange of ideas among several different fields of computer science and technology. A joint project of the above mentioned two faculties *Mathematical and Engineering Approaches to Developing Reliable and Secure Concurrent and Distributed Computer Systems*, financially supported by the Czech Science Foundation, provides the necessary funding for organization of the current series of MEMICS workshops.

Submissions are traditionally invited in the following areas: computer security; software and hardware dependability; parallel and distributed computing; formal analysis and verification; simulation; testing and diagnostics; GRID computing; computer networks; modern hardware and its design; non-traditional computing architectures; quantum computing; as well as all areas of theoretical computer science underlying the previously mentioned subjects. Moreover, this year, the scope of MEMICS was extended towards computer graphics and vision, signal and image processing, text and speech processing, human-computer interaction, especially when related with security or parallel or distributed processing.

The scientific programme of the MEMICS'10 workshop consisted of 23 contributed papers selected by the international Programme Committee from a total of 37 contributions. Each paper was reviewed by three independent reviewers who provided not only a recommendation to the Program Committee, but also gave an extensive feedback to the authors. The contributed papers were complemented by *presentations*, having the form of summaries of PhD student's works that already underwent a rigorous peer review process and have been presented at some high quality international conference. A total of 19 presentations selected from 23 submissions was also included in the programme.

All the presentations were given by PhD students who had the opportunity to speak in front of their peers and to receive immediate feedback from participating senior faculty members, including the invited lecturers. From the 23 contributed papers, 17 were selected to appear in the OASIcs proceedings. The selection was based on the Programme Committee reviews and also on the quality of presentations given during the MEMICS workshop.

The *Best paper award* came to three contributed papers, selected by the Programme Committee chairs at the end of the workshop. The awarded papers were: (i) Thomas Reinbacher, Joerg Brauer, Martin Horauer, Andreas Steininger, and Stefan Kowalewski: *Test-Case Generation for Embedded Binary Code Using Abstract Interpretation*, (ii) Martin Maška, Pavel Matula, and Michal Kozubek: *Simultaneous Tracking of Multiple Objects Using Fast Level Set-Like Algorithm*, and (iii) Zdeněk Přikryl, Jakub Kroustek, Tomáš Hruška, and Dušan Kolář: *Fast Translated Simulation of ASIPs*. The awards consisted of diplomas for all authors of the selected papers complemented with financial rewards, covered by the sponsoring organizations Red Hat Czech Republic, Zoner Software, and Honeywell Czech Republic.

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The MEMICS'10 workshop was financially supported by the Doctoral Grant 102/09/H042 from the Czech Science Foundation. Related direct and indirect support and help from the organizing faculties is also highly appreciated.

Brno, February 2011

Luděk Matyska, Tomáš Vojnar, Michal Kozubek,
and Pavel Zemčík
PC chairs of MEMICS'10

CUDA Accelerated LTL Model Checking – Revisited*

Petr Bauch[1] and Milan Češka[2]

1 **Faculty of Informatics, Masaryk University**
Brno, Czech Republic
xbauch@fi.muni.cz

2 **Faculty of Informatics, Masaryk University**
Brno, Czech Republic
xceska@fi.muni.cz

Abstract

Recently, the massively parallel architecture has been used to significantly accelerate many computation demanding tasks. For example, in [2, 5] we have shown how CUDA technology can be employed to accelerate the process of Linear Temporal Logic (LTL) Model Checking. In this paper we redesign the One-Way-Catch-Them-Young (OWCTY) algorithm [7] in order to devise a new CUDA accelerated OWCTY algorithm that will significantly outperform the original CUDA accelerated algorithm and will be resistant to slowdown caused by improper ordering of the input data representation.

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.1

1 Introduction

Model checking [1] is a wide-spread technique for automated formal verification of parallel and distributed software and hardware systems. For a given formal description of a system and desired system property, the goal of the model checking procedure is to analyse reachable system configurations in order to decide whether the system satisfies the property or not. The model checking technique generally suffers from the so called *state space explosion problem* that makes wide gap between the complexity of systems the current model checking tools can handle and the complexity of systems built in practice. As a result, the applicability of the model checking method to large industrial systems is rather limited.

A possible way to reduce the delay due to the formal verification process is to accelerate computation of verification tools using contemporary parallel hardware. Hardware platforms such as multi-core multi-CPU systems or many-core hardware accelerators have recently received a lot of attention in this aspect. At the leading edge of this class of massively parallel chip architectures are the modern Graphics Processing Units (GPU). GPUs have emerged as a revolutionary technological opportunity due to their tremendous massive parallelism, floating point capability, low cost, and ubiquitous presence in commodity computer systems.

Many key computational kernels have been redesigned to exploit the performance of this modern hardware. The key to effective utilisation of GPUs for scientific computing is the design and implementation of efficient data-parallel algorithms that can scale to hundreds of tightly coupled processing units.

In this paper we target LTL model checking, where the property to be verified is given as a formula of Linear Temporal Logic (LTL). The problem of LTL model checking can be

* This work has been supported in part by the Czech Grant Agency grants No. 201/09/1389 and 102/09/H042.

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 1–8

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

reduced to the problem of detection of an accepting cycle (cycle containing vertex denoted as accepting) in a directed graph. In our previous work [5] we have redesigned the maximal accepting predecessors (MAP) algorithm [6] for detection of an accepting cycle in terms of matrix-vector product in order to accelerate LTL model checking on many-core GPU platforms. Our experiments demonstrate that using the NVIDIA CUDA technology results in a significant speedup of verification process. The proposed method exhibits two weaknesses. First, it is the very expensive phase of preparation of data structures for consecutive CUDA processing, and second, the limited size of the state space that can fit the memory of a single CUDA device.

Further we have shown [2] that the expensive phase of encoding the state space into the appropriate representation can be itself accelerated by means of multi-core parallel processing followed by a few CUDA operations and second, we have shown how to employ multiple CUDA devices to overcome the memory limitations of a single device. Although preserving a decent efficiency of our inter-CUDA communication intensive parallel algorithm for LTL model checking, the proposed methods may affect the ordering of the representation which subsequently causes significant slowdown of the overall CUDA computation of the MAP algorithm.

In this paper we redesign the One-Way-Catch-Them-Young (OWCTY) algorithm [7] in order to devise a new CUDA accelerated OWCTY algorithm, both superior to the previous MAP algorithm in speed and robust to improper ordering in the representation.

2 Preliminaries

2.1 LTL Model Checking

To answer an LTL model checking question, the model checking tools, such as `SPIN` [9] or `DiVinE` [3], employ the automata-theoretic approach to LTL model checking, which allows to reduce the LTL model checking problem to the problem of non-emptiness of Büchi automata. In particular, the model of a system S is viewed as a finite automaton A_S describing all possible behaviours of the system. The property to be checked (LTL formula φ) is negated and translated into Büchi automaton $A_{\neg\varphi}$ describing all the behaviours violating φ. In order to check whether the system violates φ, a synchronous product $A_S \times A_{\neg\varphi}$ of A_S and $A_{\neg\varphi}$ is constructed describing those behaviours of the system that violates φ, i.e. $L(A_S \times A_{\neg\varphi}) = L(A_s) \cap L(A_{\neg\varphi})$. The automata A_S, $A_{\neg\varphi}$, and $A_S \times A_{\neg\varphi}$ are referred to as *system*, *property*, and *product* automata, respectively. System S satisfies formula φ if and only if the language of the product automaton is empty, which is if and only if there is no reachable accepting cycle in the underlying graph of the product automaton. The LTL model checking problem is thus reduced to the problem of the detection of an accepting cycle in the product automaton graph.

There are several parallel algorithms for accepting cycle detection. In [5] we have adapted the MAP algorithm [6] to allow for CUDA accelerated LTL model checking. The main idea behind this algorithm is based on the fact that each accepting vertex lying on an accepting cycle is its own predecessor. The algorithm computes a single representative accepting predecessor for each vertex. We presuppose a linear ordering $<$ of vertices (given e.g. by their memory representation) and choose the maximal accepting predecessor. If a vertex is its own maximal accepting predecessor the presence of an accepting cycle is guaranteed. If there is an accepting cycle in the graph, but none of the vertices is its own maximal accepting successor, then the maximal accepting predecessor of all the vertices of the cycle must be the same, must lie outside the cycle and can thus be marked as non-accepting. The

algorithm iteratively computes the maximal accepting predecessor for all the vertices until an accepting cycle is found or the set of accepting vertices becomes empty.

Another parallel algorithm for accepting cycle detection is One-Way-Catch-Them-Young (OWCTY) algorithm [7]. The key idea of the algorithm is maintaining an approximating set of states that may lie on an accepting cycle in the graph G. The algorithm repeatedly refines the approximating set by locating and removing states that cannot lie on any accepting cycle. The algorithm employs two rules to remove vertices from the approximating set: 1. vertices not reachable from any accepting vertex (vertices in the set F) and 2. vertices having zero in-degree.

The basic scheme of the OWCTY algorithm is given in Algorithm 1. The function Reachability(S) computes the set of all vertices that are reachable from the set S. The function Elimination(S) successively eliminates those vertices that have zero in-degree. The assignment on line 5 removes from the graph the vertices according to the 1. rule. The assignment on line 6 removes from the graph the vertices according to the 2. rule. The `while` loop terminates when fixpoint of the approximating set is reached. In the case that the approximating set is nonempty the presence of an accepting cycle is guaranteed. Moreover, we can weaken the termination condition in the following way:

Algorithm 1 *OWCTY*

proc OWCTY($G = (V, E), F \subseteq V, init\_state \in V$)

1: $S \leftarrow$ Reachability($init\_state$)
2: $old \leftarrow \emptyset$
3: **while** $S \neq old$ **do**
4: $\quad old \leftarrow S$
5: $\quad S \leftarrow$ Reachability($S \cap F$)
6: $\quad S \leftarrow$ Elimination(S)
7: **end while**
8: **return** $S \neq \emptyset$

▶ Proposition 1. Elimination(S) $= S$ is a correct termination condition of Algorithm 1.

Proof. Let us assume that $S' :=$ Reachability($S \cap F$) $=$ Elimination(S) and let $\rightsquigarrow$ denote reachability relation. Then if $S' \neq \emptyset$ we have: 1) $\forall u \in S'.\exists v \in F : u \rightsquigarrow v$, 2) $\forall v \in S'.\exists u \in S' : (u, v) \in E$. Hence there is an infinite sequence $\pi := u_1, v_1, u_2, v_2, \ldots : u_i \in F, (v_i, u_i) \in E, u_i \rightsquigarrow v_{i-1}$. And since F is finite, we may conclude that π contains an accepting cycle. ◀

2.2 CUDA Architecture

The Compute Unified Device Architectures (CUDA) [8], developed by NVIDIA, is a parallel programming model and software environment providing general purpose programming on Graphics Processing Units. At the hardware level, GPU device is a collection of multiprocessors each consisting of eight scalar processor cores, instruction unit, on-chip shared memory, and texture and constant memory caches. Every core has a large set of local 32-bit registers but no cache. The multiprocessors follow the SIMD architecture, i.e. they concurrently execute the same program instruction on different data. Communication among multiprocessors is realised through the shared device memory that is accessible for every processor core.

On the software side, the CUDA programming model extends the standard C/C++ programming language with a set of parallel programming supporting primitives. A CUDA

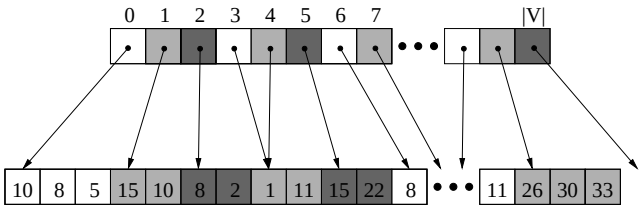


Figure 1 Adjacency list representation: $G = (V, E)$ is stored as two arrays of sizes $|V| + 1$ and $|E|$.

program consists of a *host* code running on the CPU and a *device* code running on the GPU. The device code is structured into so called *kernels*. A kernel executes the same scalar sequential program in many *data independent parallel threads*.

Each multiprocessor has several fine-grain hardware thread contexts, and at any given moment, a group of threads called a *warp* execute on the multiprocessors in a lock-step manner. When several warps are scheduled on multiprocessors, memory latencies and pipeline stalls are hidden primarily by switching to another warp.

2.3 CUDA Accelerated MAP Algorithm

To realise efficiently any CUDA-aware graph algorithm needs the graph to be represented in a compact, preferably vector-like, fashion. The MAP algorithm employs a variant of adjacency list representation, resembling *Compressed Sparse Row* (CSR) representation as illustrated in Figure 1. See [5] for more details. The key idea of the acceleration of the MAP algorithm lies in the parallel computation of the maximal accepting predecessor for all the vertices. We have devised a CUDA kernel that updates the values of the maximal accepting predecessors along the corresponding outgoing edges simultaneously for all vertices in the graph. See [5] for more details. Besides the data structure for representing the graph, the CUDA algorithm has to maintain another data structure to store the MAP values – a vector. Data manipulation thus resembles a sparse matrix (graph) vector (values of maximal accepting predecessor) multiplication pattern, which is known to be convenient for CUDA acceleration.

Our CUDA accelerated approach to LTL model checking exhibited certain weaknesses as already mentioned in [5]. Among other aspects it was the costly preparation of data structures for consecutive CUDA processing. Though we have diminished the size of this problem considerably by means of multi-core parallelisation [2], a new flaw consequently emerged. The altered ordering in the CSR representation has shown less efficient for the MAP algorithms. To understand why, we should point out that we are actually computing minimal accepting successors. Considering successors allows us to store only the forward edges and preferring smaller values inverts the BFS ordering enforced by generation (actual BFS ordering provided significantly worse results). This observation can then be explained by existence of paths going out of accepting cycles: prolonging search for maximal successor and preventing termination when one is found. While avoided by order inversion, this aspect seems to be partially restored when generation is done concurrently. The following CUDA accelerated OWCTY algorithm should prove more resistant to any improper ordering in CSR representation.

3 CUDA Accelerated OWCTY Algorithm

The non-CUDA version of OWCTY algorithm comprises of alternating execution of forward reachability and *backward elimination* (Algorithm 1). In the current context we denote elimination of vertices without immediate predecessors as backward elimination. These two operations will similarly be the building blocks of our new implementation. Their data-parallel versions to be precise.

Implementation of reachability was given sufficient space in [2] (where referred to as closure computation). We will thus in the following concentrate on describing in more detail the implementation of backward elimination and subsequently the whole OWCTY algorithm. Given the fact that the algorithm disposes of only the forward edges we were unable to follow the most obvious implementation procedure, i.e. to eliminate a vertex if all its predecessors were already eliminated. The option of providing also the backward edges would be overly complex both in time and space. Our backward elimination hence needed to consist of two steps (see Algorithm 2). The first step is performed by the CUDA kernel Progress, starting at line 7. This kernel has the purpose of propagating the property of not to be eliminated to its successors. Followed by the second kernel Check which eliminates vertices without this property. Finally, the operations Elim, SetElim, etc. are low-level bitwise operations on a piece of memory assigned to every vertex, which allows them to be performed very fast even on simple GPU processing units.

Having described the building blocks, we may proceed to the actual OWCTY algorithm implementation (see Algorithm 3). The basic layout is equivalent to the original implementation. The CUDA kernel VisAccepting sets all accepting vertices to visited. Having considered the Proposition 1, we need not to test if Reachability visited all vertices. Only its effect, the elimination of non-visited vertices is necessary (via kernel TestSet). The elimination proceeds as described above. Furthermore, if no vertex is eliminated (line 5) the algorithm terminates with resulting value stored in variable $found$. It is observable that $found$ keeps track of existence of not eliminated vertices thus providing correct answer once the main cycle terminated.

The dual version of OWCTY algorithm, here referred to as *reversed* OWCTY, may seem to present equivalent obstacles as far as the CUDA implementation is concerned. Though as stated in [2] backward reachability via forward edges is securable (with certain slowdown), allowing us to implement elimination in the trivial way as sketched above. The rest of the algorithm remains the same and the resulting efficiency of both implementation is compared in Section 5.

4 Early Termination and Combination of Algorithms

A key property of some model checking algorithms is that they can be altered to provide early termination. It means that they can detect the presence of an accepting cycle before the state space generation procedure completes its task. We were able to adapt our implementation of CUDA accelerated OWCTY algorithm to mimic this behaviour as well. The idea is very similar as in our previous papers [2, 5]. In particular, we let the CPU perform (parallel) state space generation while having the GPU apply CUDA accelerated OWCTY algorithm on partially constructed graph. If the part of the graph constructed so far contains an accepting cycle, CUDA accelerated OWCTY algorithm simply reveals it before the state space generation is complete.

To further extend the potential efficiency of the proposed model checking method we allow for both the MAP and OWCTY algorithm to be executed concurrently in the back-

Algorithm 2 *Backward Elimination*

```
1: while change do
2:   Progress(V)
3:   change, found ← false
4:   Check(V, change, found)
5:   result ← change ? true : result
6: end while

kernel Progress(V) // run in data-parallel fashion on all v ∈ V at once
7: if ¬Elim(v) then
8:   for all u ∈ Succ(v) do
9:     if ¬Elim(u) ∧ ElimPrep(u) then
10:      UnsetElimPrep(u)
11:    end if
12:  end for
13: end if

kernel Check(V, change, found) // again on all v ∈ V at once
14: if ¬Elim(v) then
15:   if ElimPrep(v) then
16:     SetElim(v)
17:     change ← true
18:   else
19:     SetElimPrep(v)
20:     found ← true
21:   end if
22: end if
```

Algorithm 3 *CUDA OWCTY*

```
1: VisAccepting(V)
2: while result do
3:   Reachability(V)
4:   TestSet(V)
5:   result ← false
6:   Elimination(V, found, result)
7: end while
8: return found
```

ground of the state space generation. This work flow, though requiring two CUDA devices, provides the best result of the two algorithms whether or not was the early termination available (and with negligible impact on their stand-alone performance).

5 Experimental Evaluation

We have implemented both variants of CUDA accelerated OWCTY algorithm as a part of DiVinE-CUDA [4]. We compared the performance of these algorithms against the original CUDA accelerated MAP algorithm [2, 5].

All the experiments were run on a Linux workstation with a quad core AMD Phenom(tm) II X4 940 Processor @ 3GHz, 8 GB DDR2 @ 1066 MHz RAM and two NVIDIA GeForce GTX 280 GPU's with 1GB of GPU memory.

Table 1 provides details on run-times of the algorithms. The total run-time includes the

	Models (seq. total time: MAP/OWCTY)	CPU cores	*CSR time*	CUDA MAP		CUDA OWCTY		CUDA OWCTY REVERSE	
				CUDA time	*total time*	*CUDA time*	*total time*	*CUDA time*	*total time*
without accepting cycle	elevator 1 (100/41)	1	24.5	6.0	31.6	0.7	26.3	0.2	**25.8**
		2	15.2	5.8	22.3	0.8	17.3	0.3	**16.8**
		3	12.1	6.1	19.2	1.2	14.3	0.3	**13.4**
	leader (697/297)	1	86.0	0.1	**87.4**	1.1	88.4	0.8	88.1
		2	49.1	4.2	54.5	2.2	52.5	1.2	**51.5**
		3	35.4	9.3	45.2	4.3	40.2	1.3	**37.2**
	peterson 1 (445/188)	1	97.9	3.5	102.3	1.0	99.8	0.5	**99.3**
		2	58.3	9.5	69.6	1.8	61.9	0.7	**60.8**
		3	41.5	10.0	52.7	2.1	44.8	0.8	**43.5**
	anderson (115/113)	1	30.6	1.5	33.2	0.5	32.2	0.2	**31.9**
		2	19.5	1.6	22.4	0.5	21.3	0.3	**21.2**
		3	15.5	4.4	21.6	1.2	18.4	0.4	**17.6**
with accepting cycle	elevator 2 (50/177)	1	27.2	0.6	28.7	1.2	29.3	0.5	**28.6**
		2	19.5	0.9	21.5	1.8	22.4	0.6	**21.2**
		3	14.6	0.9	16.4	2.0	17.5	0.5	**16.0**
	phils (397/576)	1	45.2	< 0.1	46.1	< 0.1	46.1	< 0.1	46.1
		2	29.6	< 0.1	30.3	0.1	30.4	< 0.1	30.3
		3	20.8	< 0.1	21.6	0.1	21.7	< 0.1	21.6
	peterson 2 (173/404)	1	25.7	4.0	30.5	0.4	26.9	0.3	**26.8**
		2	17.4	4.3	22.5	0.6	**18.8**	0.8	19.0
		3	12.5	0.6	**13.8**	1.2	14.4	1.0	14.2
	bakery (240/907)	1	22.1	< 0.1	**23.2**	0.4	23.6	0.2	23.4
		2	13.5	< 0.1	**14.4**	0.5	14.9	0.3	14.7
		3	6.2	< 0.1	**7.3**	0.8	8.1	0.1	7.4

Table 1 The overall run-times of the algorithms in seconds.

initialisation time (not reported in the table), CSR construction time (*CSR time*) and time spent on CUDA computation (*CUDA time*). Note that during the whole computation of the algorithm, one core oversees the communication with CUDA device and thus cannot be efficiently used in the CSR construction.

We have extended the table presented in [2] by the times for both variants of CUDA accelerated OWCTY algorithm. We can see that the reversed variant of CUDA accelerated OWCTY algorithm has better times that the standart variant. The reason behind it is that in reversed OWCTY the elimination was implemented more efficiently to the detriment of the reachability procedure. And since in most of the tested models the reachability needed considerably less iteration, it was the reversed version that thrived.

We can further see that both variants of CUDA accelerated OWCTY algorithm significantly outperform the original CUDA accelerated MAP algorithm on most valid model checking instances (without accepting cycle). Also on most of the invalid instances (with accepting cycle) the reversed OWCTY algorithm has slightly better times than the MAP algorithm. Moreover, on `peterson 2` the MAP algorithm falls behind both the OWCTY algorithms significantly. The reason is that the performance of CUDA accelerated MAP algorithm deeply depends on the ordering in CSR representation which directly affects the

number of calls to CUDA kernels [2, 5].

The improper ordering in CSR representation is even more crucial in case of multi-core acceleration of CSR representation. The parallel CSR construction usually affects the ordering and can lead to slowdown of CUDA computation as in the case of `leader`. The experiments show that the performance of the OWCTY algorithms does not depend on the ordering in CSR representation as much as the MAP algorithm. All together it seems that when the multi-core acceleration of CSR representation is utilised the reversed variant of the OWCTY algorithm is clearly a winner for CUDA computation.

6 Conclusions

We have demonstrated that the new CUDA accelerated OWCTY algorithms outperform the original MAP algorithm on valid instances of model checking problems. Moreover, the reversed variant of OWCTY algorithm has slightly better times also on invalid instances. The experiments also show that in opposite to MAP algorithm the OWCTY algorithm is resistant to improper ordering in CSR representation. This is particularly important when the order affecting multi-core acceleration of data preparation is applied. In the future we would like to include also the state space generation in CUDA acceleration thus allowing the whole model checking procedure to fully utilise the parallel potential of many-core architectures.

References

1 Christel Baier and Joost P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

2 J. Barnat, P. Bauch, L. Brim, and M. Češka. Employing Multiple CUDA Devices to Accelerate LTL Model Checking. In *16th International Conference on Parallel and Distributed Systems (ICPADS 2010)*, pages 259–266. IEEE Computer Society, 2010.

3 J. Barnat, L. Brim, and P. Ročkai. DiVinE Multi-Core – A Parallel LTL Model-Checker. In *Automated Technology for Verification and Analysis (ATVA 2008)*, volume 5311 of *LNCS*, pages 234–239. Springer, 2008.

4 J. Barnat, L. Brim, and M. Češka. DiVinE-CUDA: A Tool for GPU Accelerated LTL Model Checking. *EPTCS (PDMC 2009)*, 14:107–111, 2009.

5 J. Barnat, L. Brim, M. Češka, and T. Lamr. CUDA accelerated LTL Model Checking. In *15th International Conference on Parallel and Distributed Systems (ICPADS 2009)*, pages 34–41. IEEE Computer Society, 2009.

6 L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In *Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 352–366. Springer, 2004.

7 I. Černá and R. Pelánek. Distributed Explicit Fair Cycle Detection (Set Based Approach). In *Model Checking Software (SPIN'03)*, volume 2648 of *LNCS*, pages 49–73. Springer, 2003.

8 NVIDIA CUDA Compute Unified Device Architecture - Programming Guide Version 2.0,. `http://www.nvidia.com/object/cuda_develop.html`, June 2009.

9 G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

Process Algebra for Modal Transition Systemses*

Nikola Beneš[1] and Jan Křetínský[1,2]

1 **Faculty of Informatics, Masaryk University**
Botanická 68a, 602 00 Brno, Czech Republic
xbenes3@fi.muni.cz, jan.kretinsky@fi.muni.cz
2 **Institut für Informatik, TU München**
Boltzmannstr. 3, D-85748, Garching, Germany

Abstract

The formalism of modal transition systems (MTS) is a well established framework for systems specification as well as abstract interpretation. Nevertheless, due to incapability to capture some useful features, various extensions have been studied, such as e.g. mixed transition systems or disjunctive MTS. Thus a need to compare them has emerged. Therefore, we introduce transition system with obligations as a general model encompassing all the aforementioned models, and equip it with a process algebra description. Using these instruments, we then compare the previously studied subclasses and characterize their relationships.

Keywords and phrases modal transition systems, process algebra, specification

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.9

1 Introduction

Design and verification of parallel systems is a difficult task for several reasons. Firstly, a system usually consists of a number of components working in parallel. Component based design thus receives much attention and *composition* is a crucial element to be supported in every reasonable specification framework for parallel systems. Secondly, the behaviour of the components themselves is not trivial. One thus begins the design process with an underspecified system where some behaviour is already prescribed and some may or may not be present. The specification is then successively refined until a real implementation is obtained, where all details of the behaviour are settled. Therefore, a need for support of *stepwise refinement* design arises. This is indispensable, either due to incapability of capturing all the required behaviour in the early design phase, or due to leaving a bunch of possibilities for the implementations, such as in e.g. product lines [6]. Modal transition systems is a framework supporting both these fundamental features.

Modal transition systems (MTS) is a specification formalism introduced by Larsen and Thomsen [7, 1] allowing both for stepwise refinement design of systems and their composition. A considerable attention has been recently paid to MTS due to many applications, e.g. component-based software development [9], interface theories [10], or modal abstractions and program analysis [5], to name just a few.

The MTS formalism is based on transparent and simple to understand model of *labelled transition systems* (LTS). While LTS has only one labelled transition relation between the states determining the behaviour of the system, MTS as a specification formalism is equipped with two types of transitions: the *must* transitions capture the required behaviour, which

* The word "Systemses" in the title is deliberate. Modal transition systems is a formalism. We consider several formalisms based on modal transition systems here.

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 9–18

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

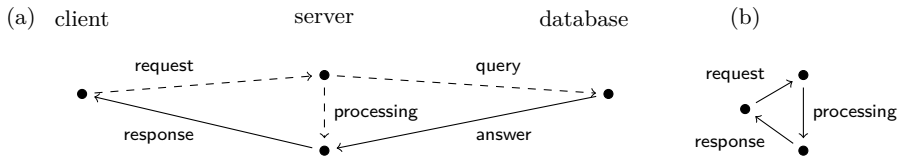


Figure 1 An example of (a) a modal transition system (b) its implementation

is present in all its implementations; the *may* transitions capture the allowed behaviour, which need not be present in all implementations. Such a system can be refined in two ways: a may transition is either implemented (and becomes a must transition) or omitted (and disappears as a transition). Figure 1 depicts an MTS that has arisen as a composition of three systems and specifies the following. A request from a client may arrive. Then we can process it directly or make a query to a database where we are guaranteed an answer. In both cases we send a response. On the right there is an implementation of the system where the processing branch is implemented and the database query branch is omitted. Note that in this formalism we can easily compose implementations as well as specifications.

While specifying may transitions brings guarantees on safety, liveness can be guaranteed to some extent using must transitions. Nevertheless, at an early stage of design we may not know which of several possible different ways to implement a particular functionality will later be chosen, although we know at least one of them has to be present. We want to specify e.g. that either processing or query will be implemented, otherwise we have no guarantee on receiving response eventually. Therefore, several formalisms extending MTS have been introduced. *Disjunctive modal transition systems* (DMTS) do not enforce a particular transition, but specify a whole set of transitions at least one of which must be present. (In our example, it would be the set consisting of processing and query transitions.) DMTS have been introduced in several flavours [8, 4, 2]. Another extension guaranteeing more structured requirements on the behaviour are *mixed transition systems* (MixTS) [3]. Here the required behaviour is not automatically allowed (not all must transitions are necessarily also may transitions) and it must be realized using other allowed behaviour. This corresponds to the situation where a new requirement can be implemented using some reused components. Moreover, it allows for some liveness properties as well. All in all, a need for more structured requirements has emerged. Therefore, we want to compare these formalisms and their expressive power.

We introduce *transition system with obligations* (OTS), a framework that encompasses all the aforementioned systems. Further, we introduce a new process algebra, since there was none for any of the discussed classes of systems. The algebra comes with the respective structural operational semantics, and thus enriches the ways to reason about all these systems. More importantly it allows us to obtain their alternative characterization and provide a more compact description language for them. Altogether, these two new tools allow us to compare all the variants of MTS and we indeed show interesting relationships among the discussed systems. We characterize the process algebra fragments corresponding to the various subclasses of OTS, such as MTS, MixTS or variants of DMTS. Since bisimulation is a congruence w.r.t. all operators of the algebra, this allows for modular analysis of the systems and also for practical optimizations based on minimization by bisimulation quotienting. Finally, since OTS allow to specify requirements in quite a general form, we can perform some important optimizations in the composition of systems. E.g., when composing DMTS we can avoid an additional exponential blowup that was unavoidable so far.

2 Preliminaries

In order to define the framework we will work in, we need a tool to handle complex requirements imposed on the systems. For this we use positive boolean formulae.

▶ **Definition 1.** A *positive boolean formula* over set X of atomic propositions is given by the following syntax:

$$\varphi ::= x \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathbf{tt} \mid \mathbf{ff}$$

where x ranges over X. The set of all positive boolean formulae over X is denoted as $\mathcal{B}^+(X)$. The semantics $[\![\varphi]\!]$ of a positive boolean formula φ is a set of subsets of X satisfying φ. It is inductively defined as follows:

$$[\![x]\!] = \{Y \subseteq X \mid x \in Y\} \quad [\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!] \quad [\![\mathbf{tt}]\!] = 2^X \quad [\![\mathbf{ff}]\!] = \emptyset \quad [\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$$

Every positive boolean formula can be uniquely represented in conjunctive normal form (CNF). It can also be uniquely represented in disjunctive normal form (DNF). In the disjunctive normal form of φ, the disjuncts are precisely the minimal elements of $[\![\varphi]\!]$ (with set inclusion). The formulae $\mathbf{tt}$ and $\mathbf{ff}$ are never needed as proper subformulae of any other formula.

We now proceed with the definition of the systems that are general enough to capture features of all the systems that we discuss in the paper.

▶ **Definition 2.** A *transition system with obligations* (OTS) over an action alphabet Σ is a triple $(\mathcal{P}, \dashrightarrow, \mathbf{\Omega})$, where P is a set of *processes*, $\dashrightarrow \subseteq P \times \Sigma \times P$ is the *may* transition relation and $\mathbf{\Omega}\colon P \to \mathcal{B}^+(\Sigma \times P)$ is the set of *obligations*.

For simplicity we also require the systems to be finitely branching, i.e. for every $P \in \mathcal{P}$ there are only finitely many $P' \in \mathcal{P}$ with $(P, a, P') \in \dashrightarrow$ for some a. Nevertheless, we could easily drop this assumption if we allowed conjunctions and disjunctions of infinite arities.

Various subclasses of OTS have been studied. We list the most important ones and depict their syntactic relationships in Fig. 2.

- A *disjunctive modal transition system* (DMTS) [8] is an OTS where the must obligations are in CNF. An arbitrary OTS can thus be expressed as a DMTS. Indeed, as noted above, any formula can be translated into CNF. However, this can cost an exponential blowup.
- A *mixed transition system* (MixTS) [3] is an OTS where the must obligations are just conjunctions of atomic predicates.

Moreover, we can impose the following *consistency requirement*

$$\mathbf{\Omega}(S) \neq \mathbf{ff} \text{ and if } \mathbf{\Omega}(S) \text{ contains } (a, T) \text{ then } S \overset{a}{\dashrightarrow} T,$$

which guarantees that all required behaviour is also allowed. This gives rise to the following systems:

- A *consistent DMTS* (cDMTS) [2] is a DMTS satisfying the consistency requirement.
- A *modal transition system* (MTS) [7] is a MixTS satisfying the consistency requirement.
- A *labelled transition system* (LTS) is an MTS such that whenever $S \overset{a}{\dashrightarrow} T$ then $\mathbf{\Omega}(S) = (a, T) \wedge \varphi$ for some φ. Since all behaviour of an LTS is both allowed and required at the same time, we also call LTS an *implementation*.

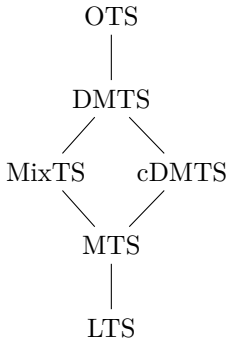


Figure 2 The syntactic hierarchy of MTS extensions

In order to define the refinement relation on the systems, we need the following auxiliary notion of refinement on formulae motivated by the following example.

▶ **Example 3.** Let us assume formulae $\varphi = (a \wedge b) \vee c$ and $\psi = A \vee C \vee D$. The renaming $R: \ a = A, c = C$ then guarantees that $\varphi \Rightarrow \psi$. This logical refinement (entailment) up to renaming is formalized in the following definition.

▶ **Definition 4.** Let $R \subseteq X \times X$, let $\varphi, \psi \in \mathcal{B}^+(X)$. We write $\varphi \sqsubseteq_R \psi$ to denote

$$\forall M \in [\![\varphi]\!] \ \exists N \in [\![\psi]\!] \ \forall n \in N \ \exists m \in M : (m, n) \in R$$

Note that if we take $R = id$, $\varphi \sqsubseteq_{id} \psi$ if and only if $\varphi \Rightarrow \psi$ (i.e. $[\![\varphi]\!] \subseteq [\![\psi]\!]$). Before proceeding to the fundamental definition of OTS, we prove the following lemmata that will be useful in later proofs. The first lemma is straightforward.

▶ **Lemma 5.** *Let $\varphi \in \mathcal{B}^+(X)$. Then $[\![\varphi]\!]$ is an upwards closed set in $(2^X, \subseteq)$.*

For the two following lemmata, assume this situation: Let X be an arbitrary set and let Y_x be an arbitrary finite set for all $x \in X$. Let $\varphi \in \mathcal{B}^+(X)$ and $\widehat{\varphi}$ be the formula that is created from φ by replacing all occurrences of x by $\bigvee Y_x$ (where $\bigvee \emptyset = \mathbf{ff}$).

▶ **Lemma 6.** *Let $Z \subseteq X$ and let $Z' \subseteq \bigcup_{z \in Z} Y_z$ such that for all $z \in Z$, there is some $y \in Y_z \cap Z'$. Then $Z \in [\![\varphi]\!]$ implies $Z' \in [\![\widehat{\varphi}]\!]$.*

Proof. The proof is done by induction on φ.

- The cases of $\varphi = \mathbf{tt}$ and $\varphi = \mathbf{ff}$ are trivial.
- $\varphi = x$. Then $\widehat{\varphi} = \bigvee Y_x$. $Z \in [\![\varphi]\!]$ implies $x \in Z$ and thus there is some $y \in Y_x \cap Z'$. Therefore $Z' \in [\![\widehat{\varphi}]\!]$ as $[\![\widehat{\varphi}]\!]$ contains $\{y\}$ and it is an upwards closed set.
- $\varphi = \psi \wedge \xi$, then $\widehat{\varphi} = \widehat{\psi} \wedge \widehat{\xi}$. Let $Z \in [\![\varphi]\!] = [\![\psi]\!] \cap [\![\xi]\!]$. Then $Z \in [\![\psi]\!]$ and $Z \in [\![\xi]\!]$. Due to the induction hypothesis, $Z' \in [\![\widehat{\psi}]\!]$ and $Z' \in [\![\widehat{\xi}]\!]$, thus also $Z' \in [\![\widehat{\psi} \wedge \widehat{\xi}]\!] = [\![\widehat{\varphi}]\!]$.
- The case of $\vee$ is similar to the previous case. ◀

▶ **Lemma 7.** *Let $Z' \subseteq \bigcup_x Y_x$ and let $Z = \{x \mid \exists y \in Y_x \cap Z'\}$. Then $Z' \in [\![\widehat{\varphi}]\!]$ implies $Z \in [\![\varphi]\!]$.*

Proof. The proof is done by induction on φ.

- The cases of $\varphi = \mathbf{tt}$ and $\varphi = \mathbf{ff}$ are trivial.
- $\varphi = x$. Then $\widehat{\varphi} = \bigvee Y_x$. As $Z' \in [\![\widehat{\varphi}]\!]$, there has to be some $y \in Y_x \cap Z'$. Thus $x \in Z$, which means that $Z \in [\![\varphi]\!]$.

- The cases of $\wedge$ and $\vee$ are similar to the proof of the previous lemma. ◀

We can now proceed to the fundamental definition of refinement of OTS.

▶ **Definition 8.** Let $(\mathcal{P}_1, \dashrightarrow_1, \mathbf{\Omega}_1)$, $(\mathcal{P}_2, \dashrightarrow_2, \mathbf{\Omega}_2)$ be two OTS and $R \subseteq \mathcal{P}_1 \times \mathcal{P}_2$. We say that R is a *refinement relation*, if $(S,T) \in R$ implies that:

- Whenever $S \overset{a}{\dashrightarrow} S'$ there is $T \overset{a}{\dashrightarrow} T'$ such that $(S',T') \in R$.
- $\mathbf{\Omega}_1(S) \sqsubseteq_{\Sigma R} \mathbf{\Omega}_2(T)$ where $\Sigma R = \{((a,S),(a,T)) \mid a \in \Sigma, (S,T) \in R\}$.

We say that S refines T (denoted as $S \leq T$) if there is a refinement relation R such that $(S,T) \in R$. Further, we say that a process I is an *implementation of a process* S if I is an implementation and $I \leq S$. We denote the set of all implementations of S by $[\![S]\!] = \{I \mid I \leq S, I \text{ is an implementation}\}$.

▶ Remark. Clearly, our definition of refinement coincides with modal refinements on all discussed subclasses of OTS.

One can easily see that every system satisfying the consistency requirement has an implementation, whereas DMTS and MixTS do not necessarily have one. We can compare various flavours of modal transition systems according to expressivity. Due to previous observation, we only consider nonempty sets of implementations.

▶ **Definition 9.** Let $\mathcal{C}, \mathcal{D}$ be subclasses of OTS. We say that $\mathcal{D}$ is at least as expressive as $\mathcal{C}$, written $\mathcal{C} \preceq \mathcal{D}$, if for every $C \in \mathcal{C}$ with $[\![C]\!] \neq \emptyset$ there is $D \in \mathcal{D}$ such that $[\![D]\!] = [\![C]\!]$.We write $\mathcal{C} \equiv \mathcal{D}$ to indicate $\mathcal{C} \preceq \mathcal{D}$ and $\mathcal{C} \succeq \mathcal{D}$, and $\mathcal{C} \prec \mathcal{D}$ to indicate $\mathcal{C} \preceq \mathcal{D}$ and not $\mathcal{C} \equiv \mathcal{D}$.

3 Process Algebra for DMTS

In this section we define a process algebra for OTS. However, since the processes represent sets of implemented systems (i.e. sets of sets of behaviours), we still need the obligation function to fully capture them. For the sake of simplicity, we introduce the parallel composition operator only in the following subsection.

▶ **Definition 10.** Let $\mathcal{X}$ be a set of process names. A *term of process algebra for OTS* is given by the following syntax:

$$P ::= \text{nil} \mid \text{co-nil} \mid a.P \mid X \mid P \wedge P \mid P \vee P \mid \lightning P$$

where X ranges over $\mathcal{X}$ and every $X \in \mathcal{X}$ is assigned a defining equality of the form $X := P$ where P is a term. The semantics is given by the following structural operational semantics rules:

$$\frac{}{a.P \overset{a}{\dashrightarrow} P} \qquad \frac{P \overset{a}{\dashrightarrow} P'}{X \overset{a}{\dashrightarrow} P'} X := P \qquad \frac{P \overset{a}{\dashrightarrow} P'}{P \wedge Q \overset{a}{\dashrightarrow} P'} \qquad \frac{P \overset{a}{\dashrightarrow} P'}{P \vee Q \overset{a}{\dashrightarrow} P'}$$

The obligation function on terms is defined structurally as follows:

$$\begin{array}{lll} \mathbf{\Omega}(\text{nil}) & = \mathbf{tt} & \mathbf{\Omega}(P \wedge Q) = \mathbf{\Omega}(P) \wedge \mathbf{\Omega}(Q) \\ \mathbf{\Omega}(\text{co-nil}) & = \mathbf{ff} & \mathbf{\Omega}(P \vee Q) = \mathbf{\Omega}(P) \vee \mathbf{\Omega}(Q) \\ \mathbf{\Omega}(a.P) & = (a,P) & \mathbf{\Omega}(\lightning P) = \mathbf{\Omega}(P) \\ \mathbf{\Omega}(X) & = \mathbf{\Omega}(P) \quad \text{for } X := P & \end{array}$$

As a convenient shortcut we introduce $?P \equiv (P \vee \text{nil})$ to capture the may transitions, i.e. an allowed behaviour that is not necessarily forced. Hence we easily obtain the following using the rules above:

$$\frac{P \overset{a}{\dashrightarrow} P'}{?P \overset{a}{\dashrightarrow} P'} \qquad \mathbf{\Omega}(?P) = \mathbf{tt}$$

We now obtain the discussed subclasses of OTS as syntactic subclasses generated by the following syntax equations (modulo transformation to CNF):

DMTS	$P ::= \text{nil} \mid a.P \mid X \mid P \wedge P \mid P \vee P \mid \not\downarrow P \mid \text{co-nil}$
cDMTS	$P ::= \text{nil} \mid a.P \mid X \mid P \wedge P \mid P \vee P$
MixTS	$P ::= \text{nil} \mid a.P \mid X \mid P \wedge P \mid P \vee \text{nil} \mid \not\downarrow P \mid \text{co-nil}$
MTS	$P ::= \text{nil} \mid a.P \mid X \mid P \wedge P \mid P \vee \text{nil}$
LTS	$P ::= \text{nil} \mid a.P \mid X \mid P \wedge P$

3.1 Composition

We define the composition operator based on synchronous message passing, as it encompasses the synchronous product as well as interleaving.

▶ **Definition 11.** Let $\Gamma \subseteq \Sigma$ be a *synchronizing alphabet*. For processes S_1 and S_2 we define the process $S_1 \parallel S_2$ as follows.

$$\frac{S_1 \overset{a}{\dashrightarrow} S_1' \quad S_2 \overset{a}{\dashrightarrow} S_2'}{S_1 \parallel S_2 \overset{a}{\dashrightarrow} S_1' \parallel S_2'}\ a \in \Gamma$$

$$\frac{S_1 \overset{a}{\dashrightarrow} S_1'}{S_1 \parallel S_2 \overset{a}{\dashrightarrow} S_1' \parallel S_2}\ a \in \Sigma \setminus \Gamma \qquad \frac{S_2 \overset{a}{\dashrightarrow} S_2'}{S_1 \parallel S_2 \overset{a}{\dashrightarrow} S_1 \parallel S_2'}\ a \in \Sigma \setminus \Gamma$$

As we may assume obligations to be in disjunctive normal form, let us denote $\mathbf{\Omega}(S_1) = \bigvee_i \bigwedge_j (a_{ij}, P_{ij})$ and $\mathbf{\Omega}(S_2) = \bigvee_k \bigwedge_\ell (b_{k\ell}, Q_{k\ell})$. We define $\mathbf{\Omega}(S_1 \parallel S_2)$ by

$$\bigvee_{i,k} \Big(\bigwedge_{j,\ell : a_{ij} = b_{k\ell} \in \Gamma} (a_{ij}, P_{ij} \parallel Q_{kl}) \wedge \bigwedge_{j : a_{ij} \notin \Gamma} (a_{ij}, P_{ij} \parallel S_2) \wedge \bigwedge_{\ell : b_{k\ell} \notin \Gamma} (b_{k\ell}, S_1 \parallel Q_{kl}) \Big)$$

Intuitively, for a process S, the set $[\![\mathbf{\Omega}(S)]\!] \subseteq 2^{\Sigma \times P}$ consists of all possible choices of successors of S that realize all obligations. Composing $[\![\mathbf{\Omega}(S_1)]\!]$ and $[\![\mathbf{\Omega}(S_2)]\!]$ in the same manner as may transitions above generates $[\![\mathbf{\Omega}(S_1 \parallel S_2)]\!]$.

Note that $[\![\mathbf{\Omega}(S)]\!]$ corresponds to DNF of obligations. Nevertheless, they can also be written equivalently in the form of a set of must transitions of DMTS, which corresponds to CNF. During the design process CNF is more convenient to use, whereas the composition has to be done in DNF even for DMTS and then translated back, thus causing an exponential blowup. However, using OTS allows for only one transformation and then the compositions are done using DNF, as the result is again in DNF. As our definition extends the previous definitions on all the discussed models, this shows another use of OTS.

▶ Remark. Refinement is a precongruence with respect to all operators of the process algebra (including the composition operator). Hence, refinemental equivalence, i.e. $\leq \cap \leq^{-1}$, is a congruence.

4 Hierarchy Results

In this section, we study the relationship between the OTS subclasses and establish the following complete result:

$$\text{LTS (implementations)} \prec \text{MTS} \prec \text{MixTS} \prec \text{cDMTS} \equiv \text{DMTS (OTS)}$$

We first show that cDMTS $\equiv$ DMTS. We do that by showing that every OTS process that has an implementation can be substituted by an OTS process that satisfies the consistency requirement and has the same set of implementations. To that end, we use an auxiliary definition of a consistency relation. This definition is a slight modification of the consistency relation defined in [8]. In the definition, the notation $2^{\mathcal{P}}_{\text{Fin}}$ stands for the set of all finite subsets of $\mathcal{P}$.

▶ **Definition 12** (consistency)**.** Let $(\mathcal{P}, \dashrightarrow, \mathbf{\Omega})$ be a OTS. A subset C of $2^{\mathcal{P}}_{\text{Fin}}$ is called a *consistency relation* if for all $\{S_1, \ldots, S_n\} \in C$ and $i \in \{1, \ldots, n\}$ there is $X \in [\![\mathbf{\Omega}(S_i)]\!]$ such that for all $(a, U) \in X$ there are $S_j \overset{a}{\dashrightarrow} T_j$ (for all j) such that $\{U, T_1, \ldots, T_n\} \in C$.

It may be easily seen that an arbitrary union of consistency relations (for given OTS) is also a consistency relation. Therefore, we may talk about the greatest consistency relation. The following lemma explains the motivation behind the consistency relation, i.e. that a set of processes is consistent if it has a common implementation.

▶ **Lemma 13.** *Let $S_1, \ldots, S_n$ be processes. There exists a consistency relation C containing $\{S_1, \ldots, S_n\}$ if and only if $\bigcap_{1 \le i \le n} [\![S_i]\!] \neq \emptyset$.*

Proof. Recall that $\varphi \sqsubseteq_{\Sigma\le} \psi$ if and only if for all $M \in [\![\varphi]\!]$ there is some $N \in [\![\psi]\!]$ such that for all $(a, T) \in N$ there is some $(a, S) \in M$ such that $S \le T$.

We show that $C = \{\{S_1, \ldots, S_k\} \mid k \in \mathbb{N}, \bigcap_i [\![S_i]\!] \neq \emptyset\}$ is a consistency relation. Let $\{S_1, \ldots, S_n\} \in C$, let $I \in \bigcap_i [\![S_i]\!]$ and let $i \in \{1, \ldots, n\}$ be arbitrary. Take $M = \{(a, J) \mid I \overset{a}{\dashrightarrow} J\}$. Clearly, $M \in [\![\mathbf{\Omega}(I)]\!]$ as I is an implementation. Due to $\mathbf{\Omega}(I) \sqsubseteq_{\Sigma\le} \mathbf{\Omega}(S_i)$ there has to be some $N \in [\![\mathbf{\Omega}(S_i)]\!]$ such that for each $(a, U) \in N$ there is $(a, J) \in M$ such that $J \le U$.

Let now $X = N$ and let $(a, U) \in X$. Then $I \overset{a}{\dashrightarrow} J$ with $J \le U$. Therefore, as $I \le S_j$, $S_j \overset{a}{\dashrightarrow} T_j$ and $J \le T_j$ for all j. Thus $J \in [\![U]\!] \cap \bigcap_i [\![T_i]\!]$ and $\{U, T_1, \ldots, T_n\} \in C$.

To show the converse, assume that there is a consistency relation C containing $\{S_1, \ldots, S_n\}$. We know that for all i there is some $X \in \mathbf{\Omega}(S_i)$ such that for all $(a, U) \in X$ there are $S_j \overset{a}{\dashrightarrow} T_j$ (for all j) such that $\{U, T_1, \ldots, T_n\} \in C$. For fixed i, we denote the chosen X as X_i. We construct I coinductively as follows:

$$\mathbf{\Omega}(I) = \bigwedge_i \bigwedge_{(a,U) \in X_i} (a, J_i^U)$$

with $I \dashrightarrow$ transitions to all (a, J_i^U), where J_i^U is a common implementation of U, T_1, …, T_n with T_i given above. Clearly, I is an implementation of all S_i. ◀

We now proceed with the construction of a new consistent OTS that is equivalent to the original OTS.

▶ **Definition 14.** Let $(\mathcal{P}, \dashrightarrow, \mathbf{\Omega})$ be a OTS, Con its greatest consistency relation. We create a new OTS as $(\mathsf{Con}, \dashrightarrow, \mathbf{\Omega})$ where

- $\mathcal{S} \overset{a}{\dashrightarrow} \mathcal{T}$ whenever for all $S \in \mathcal{S}$, $S \overset{a}{\dashrightarrow} T$ with $T \in \mathcal{T}$.

- $\mathbf{\Omega}(\mathcal{S}) = \bigwedge_{S\in\mathcal{S}} \widehat{\mathbf{\Omega}(S)}$ where $\widehat{\mathbf{\Omega}(S)}$ is the formula that is created from $\mathbf{\Omega}(S)$ by replacing all occurrences of (a, U) by $\bigvee\{(a, \{U, T_1, \ldots, T_n\}) \mid \forall i : S_i \xrightarrow{a} T_i, \{U, T_1, \ldots, T_n\} \in \mathsf{Con}\}$ (where $\bigvee \emptyset = \mathbf{ff}$).

Note that due to the properties of Con, $\mathbf{\Omega}(\mathcal{S})$ is never $\mathbf{ff}$. We prove that the construction is correct, i.e. for every consistent process of the original OTS, we have indeed a process of the new OTS with the same set of implementations.

▶ **Theorem 15.** *Let S be a process. Then $[\![S]\!] \neq \emptyset$ if and only if $\{S\} \in \mathsf{Con}$. Moreover, if $\{S\} \in \mathsf{Con}$ then $[\![S]\!] = [\![\{S\}]\!]$.*

Proof. The first part of the theorem is already included in Lemma 13. We thus prove the second part. We first show that $I \in [\![S]\!]$ implies $I \in [\![\{S\}]\!]$. We define R as:

$$R = \{(I, \{S_1, \ldots, S_n\}) \mid n \in \mathbb{N}, \forall i : I \in [\![S_i]\!], \{S_1, \ldots, S_n\} \in \mathsf{Con}\}$$

and prove that R is a refinement relation. Let $(I, \{S_1, \ldots, S_n\}) \in R$.

- Let $I \xrightarrow{a} J$. Then, as $I \leq S_i$, $S_i \xrightarrow{a} T_i$ with $J \leq T_i$ for all i. Thus also $\{S_1, \ldots, S_n\} \xrightarrow{a} \{T_1, \ldots, T_n\}$ and $(J, \{T_1, \ldots, T_n\}) \in R$.
- Let $\mathbf{\Omega}(I) = \varphi$, $\mathbf{\Omega}(\{S_1, \ldots, S_n\}) = \psi$. We need to show that $\varphi \sqsubseteq_{\Sigma R} \psi$.
  Let $M \in [\![\varphi]\!]$. Then, as $I \leq S_i$ for all i, there exist $N_i \in [\![\mathbf{\Omega}(S_i)]\!]$ such that for all $(a, U) \in N_i$ exists $(a, J) \in M$ with $J \leq U$ (due to $\varphi \sqsubseteq_{\Sigma\leq} \mathbf{\Omega}(S_I)$). We use the notation $J_{(a,U)}$ to denote such J.
  Let now $N = \{(a, \{U, T_1, \ldots, T_n\}) \mid \exists i : (a, U) \in N_i, \forall j : S_j \xrightarrow{a} T_j, J_{(a,U)} \leq T_j, \{U, T_1, \ldots, T_n\} \in \mathsf{Con}\}$. Clearly, for all $(a, \{U, T_1, \ldots, T_n\}) \in N$ there is some $(a, J) \in M$ such that $(J, \{U, T_1, \ldots, T_n\}) \in R$ (we take $J = J_{(a,U)}$).
  We need to prove that $N \in [\![\psi]\!]$. In other words, we need to prove that for all i, $N \in [\![\widehat{\mathbf{\Omega}(S_i)}]\!]$. That is, however, a straightforward corollary of Lemma 6 (take $Z = N_i$, $Z' = N$).

We now show that $I \in [\![\{S\}]\!]$ implies $I \in [\![S]\!]$. We define R as:

$$R = \{(I, S) \mid I \leq \mathcal{S} \text{ with } S \in \mathcal{S} \in \mathsf{Con}\}$$

and prove that R is again a refinement relation. Let $(I, S) \in R$ and let $\mathcal{S}$ be such that $I \leq \mathcal{S}$ and $S \in \mathcal{S}$.

- Let $I \xrightarrow{a} J$. Then $\mathcal{S} \xrightarrow{a} \mathcal{T}$ with $J \leq \mathcal{T}$ and thus $S \xrightarrow{a} T$ with $T \in \mathcal{T}$. Thus also $(J, T) \in R$.
- Let $\mathbf{\Omega}(I) = \varphi$, $\mathbf{\Omega}(S) = \psi$. We need to show that $\varphi \sqsubseteq_{\Sigma R} \psi$. Let $M \in [\![\varphi]\!]$. Due to the fact that $\varphi \sqsubseteq_{\Sigma\leq} \mathbf{\Omega}\mathcal{S}$, we know that there exists $N' \in [\![\mathbf{\Omega}(\mathcal{S})]\!]$ such that for all $(a, \{U, T_1, \ldots, T_k\}) \in N'$ there exists $(a, J) \in M$ with $J \leq \{U, T_1, \ldots, T_k\}$. Take $N = \{(a, U) \mid (a, \mathcal{T}) \in N' \text{ with } U \in \mathcal{T}\}$. Clearly, as $N' \in [\![\mathbf{\Omega}(\mathcal{S})]\!]$ also $N' \in [\![\widehat{\mathbf{\Omega}(S)}]\!]$. Using Lemma 7, we get that $N \in [\![\mathbf{\Omega}(S)]\!]$ (take $Z' = N', Z = N$). ◀

The following lemma shows that MixTS $\prec$ cDMTS.

▶ **Lemma 16.** *There is no MixTS M such that $[\![M]\!] = [\![a.\mathrm{nil} \vee b.\mathrm{nil}]\!]$.*

Proof. We first note that any equation defining a MixTS may be written in the following normal form:

$$X := \bigwedge_i ?a_i.S_i \wedge \bigwedge_j \not\downarrow a_j.T_j$$

Clearly, there are three implementations of $a.\text{nil} \vee b.\text{nil}$, namely $a.\text{nil}$, $b.\text{nil}$ and $a.\text{nil} \wedge b.\text{nil}$. Let thus M have these three implementations. Clearly, the $\not\downarrow$ part of M has to be empty (i.e. $\mathbf{\Omega}(M) = \mathbf{tt}$) as M can force neither a transition nor b transition. But then $\text{nil} \in [\![M]\!]$. ◀

Due to Theorem 15, we have a syntactic characterization of consistent OTS. Since we now know MixTS $\prec$ DMTS, a question arises whether such a characterization can be obtained also for consistent MixTS. Observe that the previous construction transforms every MixTS into a consistent OTS with formulae in CNF where all literals in one clause have the same action. One might be tempted to consider the following syntactic characterization of consistent MixTS:

$$P ::= \text{nil} \mid X \mid a.P \mid P \wedge P \mid \bigvee_i a.P_i$$

However, that is not the case, as shown by the following lemma. Hence, this question remains open.

▶ **Lemma 17.** *There is no MixTS M such that $[\![M]\!] = [\![(a.(a.\text{nil} \wedge b.\text{nil}) \vee a.nil) \wedge ?a.a.\text{nil}]\!]$.*

Proof. Let $M = \bigwedge_i ?a.N_i \wedge \bigwedge_j \not\downarrow a.O_j$. (All outgoing transitions from M have to be a-transitions.) We make the following observations:

- For all j, $\mathbf{\Omega}(O_j) = \mathbf{tt}$. Otherwise, $a.\text{nil}$ could not be an implementation of M.
- Also, for all j, $O_j \xrightarrow{a}$. Otherwise, $a.(a.\text{nil} \wedge b.\text{nil})$ could not be an implementation of M.
- There has to be some k such that $a.\text{nil} \in [\![N_k]\!]$, as $a.\text{nil} \wedge a.a.\text{nil}$ also has to be an implementation of M.

We now show that $a.a.\text{nil}$ is an implementation of M. Let R' be an arbitrary refinement relation such that $(a.\text{nil}, N_k) \in R'$ (we know that such R' exists as $a.\text{nil} \leq N_k$). Take R as

$$R = id \cup R' \cup \{(a.a.\text{nil}, M)\} \cup \{(a.\text{nil}, O_j) \mid \forall j\}$$

We now show that R is a refinement.

- $a.a.\text{nil} \xrightarrow{a} a.\text{nil}$ is matched by $M \xrightarrow{a} N_k$.
- $\mathbf{\Omega}(a.a.\text{nil}) = (a, a.\text{nil})$, $\mathbf{\Omega}(M) = \bigwedge_j (a, O_j)$, thus $\mathbf{\Omega}(a.a.\text{nil}) \sqsubseteq_{\Sigma R} \mathbf{\Omega}(M)$.
- $((a.\text{nil}), N_k) \in R'$, therefore the conditions of refinement are satisfied.
- $a.\text{nil} \xrightarrow{a} \text{nil}$ is matched by $O_j \xrightarrow{a} \text{nil}$
- As $\mathbf{\Omega}(O_j) = \mathbf{tt}$, clearly $\mathbf{\Omega}(a.\text{nil}) \sqsubseteq_{\Sigma R} \mathbf{\Omega}(O_j)$.

However, $a.a.\text{nil}$ is not an implementation of $(a.(a.\text{nil} \wedge b.\text{nil}) \vee a.nil) \wedge ?a.a.\text{nil}$. ◀

Finally, we show that MTS $\prec$ MixTS.

▶ **Lemma 18.** *There is no MTS S such that $[\![S]\!] = [\![?a.b.\text{nil} \wedge ?a.c.\text{nil} \wedge \not\downarrow a.(?b.\text{nil} \wedge ?c.\text{nil})]\!]$.*

Proof. Similarly to MixTS, any equation defining a MTS can be written in the following normal form:

$$X = \bigwedge_i ?a_i.S_i \wedge \bigwedge_j a_j.T_j$$

There are three implementations which S has to possess and those are $a.b.\text{nil}$, $a.c.\text{nil}$, and $a.b.\text{nil} \wedge a.c.\text{nil}$ and S cannot possess any other implementation. Clearly, S cannot be of the form $a.T \wedge P$, as then T would have to satisfy $b.\text{nil} \leq T$ (as $a.b.\text{nil} \leq S$), also $c.\text{nil} \leq T$ (as $a.c.\text{nil} \leq S$), yet it cannot satisfy $(b.\text{nil} \wedge c.\text{nil}) \leq T$ (as $a.(b.\text{nil} \wedge c.\text{nil}) \not\leq S$). This is not possible as it can be proven that if $P \leq T$ and $Q \leq T$ then also $P \wedge Q \leq T$ for all OTS. Therefore $S = \bigvee_i ?a.S_i$ and thus $\mathbf{\Omega}(S) = \mathbf{tt}$. But then $\text{nil} \leq S$ and S has more implementations than $?a.b.\text{nil} \wedge ?a.c.\text{nil} \wedge \not\downarrow a.(?b.\text{nil} \wedge ?c.\text{nil})$. ◀

For the sake of completeness, we also state that LTS $\prec$ MTS. This trivially follows, as every LTS only has one implementation, whereas e.g. $?a.\mathrm{nil}$ has two implementations $a.\mathrm{nil}$ and nil.

5 Conclusion and Future Work

We have introduced a new formalism of transition system with obligations together with its process algebra. We have used it to compare various previously studied systems. The main result shows that general DMTS are not more powerful than consistent DMTS, whereas mixed transition systems are strictly less expressive. Furthermore, we have given an alternative syntactic characterizations of the studied systems, although a complete syntactic criterion for consistent mixed transition systems remains as a future work. Surprisingly, using more general OTS leads to some optimizations in computation of the composition that were not possible in the previously used frameworks (as discussed in [2]).

Acknowledgements Nikola Beneš has been supported by Czech Grant Agency grant no. GD102/09/H042. Jan Křetínský is a holder of Brno PhD Talent Financial Aid.

References

1 A. Antonik, M. Huth, K. G. Larsen, U. Nyman, and A. Wasowski. 20 years of modal and mixed specifications. *Bulletin of the EATCS no. 95*, pages 94–129, 2008.

2 N. Beneš, I. Černá, and J. Křetínský. Disjunctive modal transition systems and generalized LTL model checking. Technical report FIMU-RS-2010-12, Faculty of Informatics, Masaryk University, Brno, 2010.

3 Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.

4 H. Fecher and M. Steffen. Characteristic mu-calculus formulas for underspecified transition systems. *ENTCS*, 128(2):103–116, 2005.

5 M. Huth, R. Jagadeesan, and D. A. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *Proc. of ESOP'01*, volume 2028 of *LNCS*, pages 155–169. Springer, 2001.

6 K. G. Larsen, U. Nyman, and A. Wasowski. Modeling software product lines using color-blind transition systems. *STTT*, 9(5-6):471–487, 2007.

7 K. G. Larsen and B. Thomsen. A modal process logic. In *LICS*, pages 203–210. IEEE Computer Society, 1988.

8 K. G. Larsen and L. Xinxin. Equation solving using modal transition systems. In *LICS*, pages 108–117. IEEE Computer Society, 1990.

9 J.-B. Raclet. Residual for component specifications. In *Proc. of the 4th International Workshop on Formal Aspects of Component Software*, 2007.

10 J.-B. Raclet, E. Badouel, A. Benveniste, B. Caillaud, and R. Passerone. Why are modalities good for interface theories? In *ACSD*, pages 119–127. IEEE, 2009.

A Simple Topology Preserving Max-Flow Algorithm for Graph Cut Based Image Segmentation

Ondřej Daněk and Martin Maška

Centre for Biomedical Image Analysis, Faculty of Informatics
Masaryk University, Brno, Czech Republic
xdanek2@fi.muni.cz

Abstract

In this paper, we propose a modification to the Boykov-Kolmogorov maximum flow algorithm [2] in order to make the algorithm preserve the topology of an initial interface. This algorithm is being widely used in computer vision and image processing fields for its efficiency and speed when dealing with problems such as graph cut based image segmentation. Using our modification we are able to incorporate a topology prior into the algorithm allowing us to apply it in situations in which the inherent topological flexibility of graph cuts is inconvenient (e.g., biomedical image segmentation). Our approach exploits the simple point concept from digital geometry and is simpler and more straightforward to implement than previously introduced methods [14]. Due to the NP-completeness of the topology preserving problem our algorithm is only an approximation and is initialization dependent. However, promising results are demonstrated on graph cut based segmentation of both synthetic and real image data.

Keywords and phrases maximum flow, topology preserving, image segmentation, graph cuts

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.19

1 Introduction

Modern approaches to image segmentation often formulate the problem in terms of minimization of a suitable energy functional. Such methods have many benefits. Most notably, mathematically well-founded algorithms can be used to solve the originally vaguely specified task. Among the most popular energy minimization algorithms for image segmentation belong the level set framework [11] and recently also the graph cut framework [1, 3] both having their pros and cons depending on a particular situation. In this paper, we focus on the latter one.

Graph cuts, originally developed as an elegant tool for interactive object cutout, quickly emerged as a general technique for solving diverse computer vision problems such as image restoration or stereo [3]. In some sense, they can be seen as a combinatorial counterpart of the level set method [1]. Likewise level sets they are applicable to a wide range of energy functions [9], directly extensible to N-dimensional space, topologically flexible and with implicit boundary representation. Moreover, they offer integration of various types of regional or geometric constraints and the ability to reach global optima in polynomial time [2]. In this framework, the input image is converted to a weighted graph with the energy function encoded in the edge weights. Subsequently, a minimum $st-$cut [5] is found, effectively yielding a global minimum of the energy. Typically, maximum flow algorithms are used to find a minimum cut in the graph based on the Ford-Fulkerson max-flow/min-cut duality theorem [5].

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 19–25

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The aforementioned topological flexibility of graph cuts is not always desirable. There are situations in which the number of objects in the image and their topology is known in advance, e.g., in biomedical image segmentation only one object topologically equivalent to a sphere should be segmented during brain extraction. Topology is also an important prior for object tracking where objects are not allowed to split or join. The topology preserving problem has been studied extensively in the context of level sets [7]. To the best of our knowledge, the literature is not as rich in the graph cut universe with the work of Zeng et al. [14] being the only relevant. In their work, the topology preservation is embedded into the maximum flow computation. Unfortunately, the algorithm is rather complicated with description missing many important details[1]. They also showed that the topology preserving problem is NP-complete so the devised algorithm no longer guarantees to reach a global minimum. A partially similar problem is addressed also in [12]. Another option of enforcing the topology preservation is through integration of hard constraints into the energy function itself. However, this may involve considerable amount of user interaction.

In this paper, we propose a new topology preserving maximum flow algorithm for graph cut based image segmentation. Similarly to [14] our algorithm is a modification of the Boykov-Kolmogorov algorithm [2] in a way that guarantees that the output of the algorithm conforms (in the topological sense) to a given initial interface. It is achieved by making sure that the topology of the initialization is preserved during label changes throughout the computation. We borrow several ideas from [14], however, our method is simpler and generally less error-prone implementation-wise. Nevertheless, it is still an approximation, i.e., only locally optimal solution is produced. We demonstrate the potential of the proposed method on graph cut based segmentation of both synthetic and real image data using the Chan-Vese segmentation model [4, 13].

This paper is organized as follows. In Section 2 a brief review of the Boykov-Kolmogorov maximum flow algorithm and simple point concept from digital geometry is given. The proposed modifications, complexity guarantees and differences from [14] are described in Section 3. Section 4 contains experimental results of the devised algorithm. We conclude the paper in Section 5.

2 Preliminaries

2.1 The Boykov-Kolmogorov Algorithm

The maximum flow algorithm introduced by Boykov and Kolmogorov is one of the most popular when dealing with graph cut based image processing [2].

▶ **Definition 1.** Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote a directed graph with two distinguished nodes s and t, where every edge $(u, v) \in \mathcal{E}$ is assigned a non-negative real valued capacity c_{uv}. A *flow* is a mapping $f : \mathcal{E} \to \mathrm{R}^+$. It is called feasible if:

1. $\forall (u, v) \in \mathcal{E} : f_{uv} \leq c_{uv}$ (capacity constraint)
2. $\forall u \in \mathcal{V} \setminus \{s, t\} : \sum_{(u,v)\in\mathcal{E}} f_{uv} = \sum_{(w,u)\in\mathcal{E}} f_{wu}$ (flow conservation rule)

The flow value $|f|$ is defined as $\sum_{(s,v)\in\mathcal{E}} f_{sv}$ and the maximum flow problem is a problem of finding a feasible flow of a maximum value.

To find a maximum flow the Boykov-Kolmogorov algorithm (BKA) uses the augmenting path strategy [5]. This strategy involves iterative searching of a non-saturated path from s

[1] The author provided implementation has stability issues and the source code seems to perform operations not mentioned in the paper.

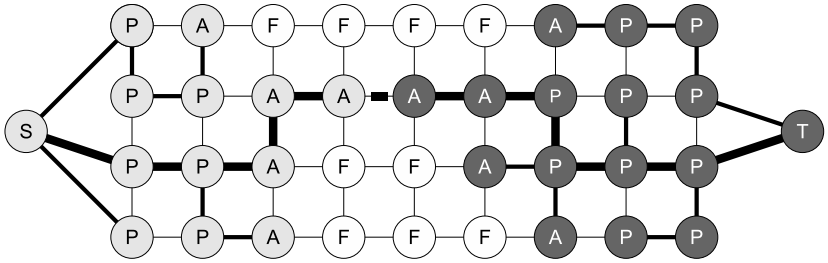


Figure 1 Boykov-Kolmogorov maximum flow algorithm scheme with active, passive and free nodes. An augmenting path (bold) is found when the two dynamic trees touch.

to t in the *residual graph* and pushing as much flow as possible along this path. A residual graph is obtained from $\mathcal{G}$ by taking the *residual capacity* $c^f_{uv} = c_{uv} - f_{uv}$ as the edge capacity for all $(u, v) \in \mathcal{E}$. The popularity and efficiency of BKA stems from the way augmenting paths are searched. It grows two dynamic trees from the terminal nodes s and t and an augmenting path is found when the two trees touch. This stage is called *tree growth*. In the *augmentation* stage the flow is sent along this path. This step may break the trees into forests (edges become saturated). Subsequently, *adoption* stage is performed to restore the two trees and the whole process is repeated. A visualization of BKA is depicted in Fig. 1. For a detailed description, please refer to [2].

Obtaining the requisite minimum cut is straightforward after the maximum flow has been found. Due to the min-cut/max-flow duality [5] the minimum cut is determined by the saturated edges (i.e., edges with zero residual capacity) and the cost of the cut is the same as the maximum flow value. In Section 3 we explain how these principles are exploited in image segmentation.

2.2 Digital Topology and Simple Point Concept

Digital topology is a subfield of digital geometry that deals with topological properties of digital (binary) images/objects, i.e., spatial properties such as connectedness (or number of objects and holes) that are invariant under certain kind of transformations (e.g., continuous deformations involving stretching, etc.) [8]. In this context, a simple point refers to a point whose switching from background to foreground or vice versa does not change the topology of the digital image. It is one of the fundamental ideas allowing topology preserving deformations of digital images. A fast characterization of simple points in 2D has been introduced by Klette and Rosenfeld [8]. Their method considers the number of connected background and foreground components in the 8-neighbourhood of the investigated point and can be efficiently implemented using a look-up table. An extension to 3D employing a breadth-first search in a small graph has been proposed in [10].

3 Topology Preserving Algorithm

In traditional graph cut based image segmentation a graph is created from the image where each node in the graph corresponds to an image voxel (plus the two terminal nodes s and t connected to all non-terminal nodes are added) and with the energy encoded in the edge weights [1]. A maximum flow algorithm is then used to find the minimum st-cut effectively

yielding a global minimum of the energy. The final node labels are determined by the minimum cut partitioning. When using BKA this is equivalent to a so-called *tree membership*, i.e., the node/pixel is implicitly labelled as background or foreground depending on whether it lies in the s or t tree, respectively, after the computation has finished.

The described method does not impose any topology constraints on the result. In general, the final segmentation may contain arbitrary number of objects, holes, etc. To avoid this (for the reasons given in the introduction) we modify BKA to handle labels and topology changes explicitly. The modifications to particular stages of BKA are presented in the following subsections.

3.1 Initialization

Node labels are initialized using a user supplied mask (interface). The algorithm ensures that the final segmentation conforms to this initialization in the topological sense, e.g., if the initial mask contains one object with a hole there will be a single object with a hole on the output. In object tracking the initial mask may typically correspond to the segmentation from the previous time point.

The algorithm starts with a zero flow. Instead of two trees, four trees are maintained during the computation. We will denote them S_F, S_B, T_F and T_B. Initially, S_F tree contains nodes labelled as foreground and connected to s through an edge of non-zero residual capacity. Similarly, T_B contains nodes labelled as background and connected to t through an edge of non-zero residual capacity. Analogously for S_B and T_F.

3.2 Tree Growth

The tree nodes are called *active* if they are on the border of the tree (the tree can grow from them) otherwise they are *passive*. Nodes that do not belong to any of the trees are *free*. See Fig. 1 for illustration. Initially, all tree nodes are active. In this stage the four trees grow by acquiring new children for their active nodes. An active node p is chosen and its neighbours connected through an edge of non-zero residual capacity are considered for growth. Let $l(p)$ denote the label of p and $t(p)$ the associated tree. Following situations may arise when considering neighbouring node q:

- q is free: If $l(p) = l(q)$ then q is recruited as a child of p. If $l(p) \neq l(q)$ and q is simple then it is also recruited as a child of p and relabelled to $l(p)$. If q is recruited it becomes active. Nothing is done otherwise.
- $t(p) \neq t(q)$: An augmenting path has been found (irrespective of the node labels). The algorithm continues with the augmentation stage.
- $t(p) = t(q)$ and $l(p) \neq l(q)$: q is recruited as a child of p if it is simple and its label is associated to the opposite tree (recall that s is associated with the background and t with the foreground), i.e., if either (1) $t(q) = t$ and $l(q) = b$ or (2) $t(q) = s$ and $l(q) = f$. If q is recruited it is relabelled to $l(p)$ and becomes active.

As soon as all neighbours of p are explored the node becomes passive and a new active node is picked. When there are no active nodes the computation ends.

3.3 Augmentation and Adoption

These two stages remain the same as in the original algorithm. During augmentation nodes behind saturated edges and all their descendants became orphans. During the adoption stage new parents are searched for the orphans among their neighbouring nodes (in the same subtree). If no admissible parent is found the node becomes free.

3.4 Active Node Selection Rules

To ensure homogeneous propagation of the segmentation boundary (as in the level set algorithms) active nodes closest to the frontier between the foreground and background should be handled first in the tree growth stage. Various methods can be used to efficiently extract nodes closest to the frontier. The bucket priority queue approach introduced in [14] is not correct according to us and caused buffer underflows in both the original and our implementation. Instead we store queues of active nodes with the same distance from the separating boundary in an associative array. This approach has a logarithmic time complexity (in the number of graph nodes), however, it is correct and in practice as efficient (in both memory and time) as the constant-time method of [14]. To initialize the distance attribute of each node an L1 metric distance transform is employed in the beginning of the computation. Subsequently, this attribute is updated whenever a node label changes.

3.5 Complexity Guarantees and Discussion

The modified algorithm still runs in polynomial time. However, note that it is no longer guaranteed to reach a global minimum of the corresponding energy functional. Similarly to the level set based algorithms [7], only a locally optimal solution is obtained that may strongly depend on the initialization. As proved in [14] minimizing the original energy with the topology preserving constraint is an NP-complete problem. The output segmentation is given by the explicit node labels handled in the algorithm (i.e., not by the final tree membership of nodes). Because simple point check is always performed before node relabelling in the tree growth stage the solution topology has to conform to the initial mask. Finally, the main difference between our algorithm and [14] is the elimination of the overly complex inter- and intra-label steps. We treat all augmenting paths equally irrespective of the labelling.

4 Experimental Results

In this section we present the results of the proposed algorithm at segmentation of both real and synthetic image data. The Chan-Vese segmentation model [4, 13] is used as the energy functional being minimized. This model aims for partitioning the input image into two possibly disconnected regions (i.e., foreground and background) minimizing their intensity variance and the length of the separating boundary. We compare the results of the topology preserving algorithm with those obtained using the conventional Boykov-Kolmogorov algorithm. Three images were used for the comparison as depicted in Fig. 2.

The first experiment consisted of brain MRI image segmentation. Undesirable results are produced using the conventional topology-free algorithm where bright parts of the image are segmented. On the other hand, a single object corresponding to the brain is extracted using the topology preserving algorithm with the depicted initialization. Fluorescently stained cell nuclei are segmented in the second experiment. Using the standard algorithm all three nuclei merge into one object. This can be avoided using the topology preserving algorithm with initialization containing exactly three seeds as illustrated on the second row in Fig. 2. In the last experiment, topology preserving algorithm is used to keep the middle and ring fingers separated in the final segmentation. On a side note, despite our algorithm is different, we are also able to reproduce the results presented in [14].

Even though the results of the traditional and topology-preserving algorithms vary we have also conducted a comparison of running times of both methods to illustrate the performance penalty incurred by the additional simple point inspections. The test was performed on

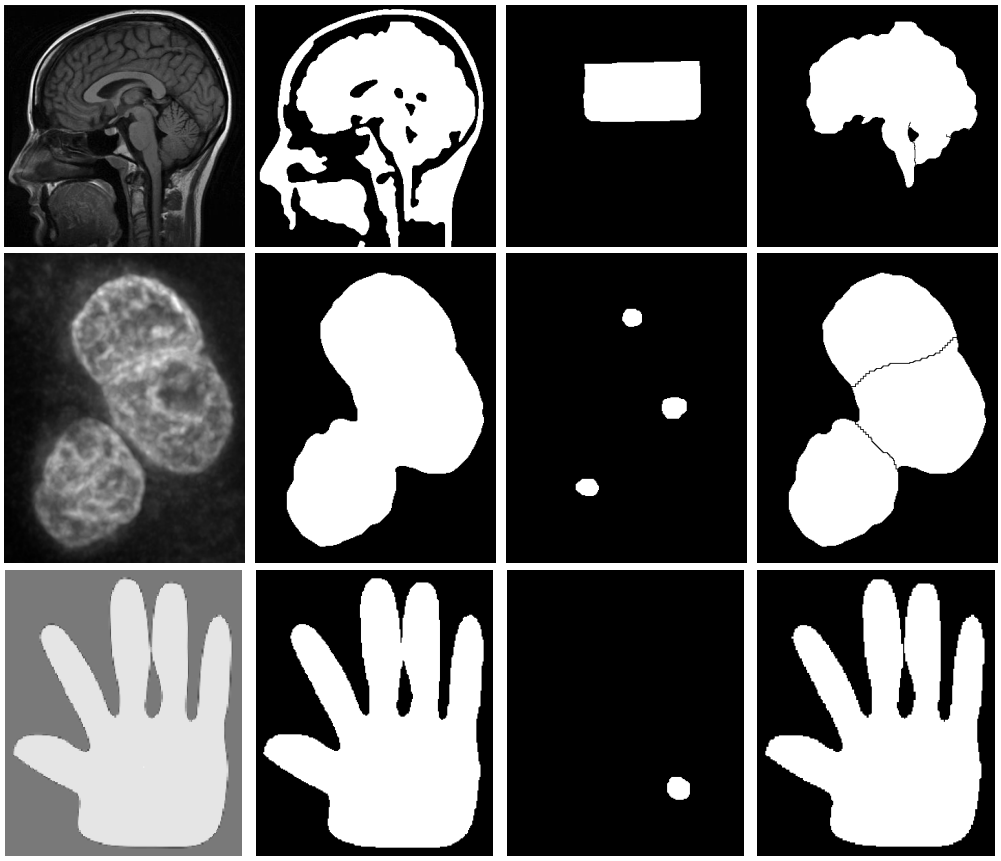


Figure 2 First column (from left): Input image. Second column: Segmentation using the conventional topology-free graph cuts. Third column: Initial mask for the topology preserving algorithm. White markers correspond to the foreground. Fourth column: Results of the topology preserving algorithm.

a common laptop equipped with an Intel Core 2 Duo processor at 2.0 GHz and 4 GB of RAM. As can be seen from the numbers listed in Table 1, the speed penalty is quite low in 2D. However, according to our experiments the topology preserving algorithm may be significantly slower in specific situations. Finally, we have not examined the performance of the algorithm in 3D. In this case a larger performance hit is to be expected due to the more complex simple point inspection routine.

5 Conclusion

A modification of the Boykov-Kolmogorov algorithm allowing topology preserving graph cut based image segmentation has been introduced in this paper. This modification is based on the simple point concept from digital geometry and is simpler than previously proposed algorithms. Despite its relative simplicity, it is able to achieve the same results and is ready for use in situations in which topology preserving is desirable. This was verified on a series of segmentation experiments. In our future work we would like to investigate the possibility of integration of topology preserving constraints also to other maximum flow algorithms such as the Push-Relabel method [6] or even the dynamic maximum flow algorithms. Implementation of the method described in this paper can be downloaded from

Table 1 Comparison of running times of the original Boykov-Kolmogorov algorithm and the proposed topology preserving modification.

Input image	Size	Boykov-Kolmogorov	Topology-preserving
Brain	350×350	2.02 s	2.89 s
Cell nuclei	280×361	0.49 s	0.51 s
Hand	228×275	0.10 s	0.13 s

our website *http://cbia.fi.muni.cz/projects/graph-cut-library.html*.

Acknowledgements This work has been supported by the Ministry of Education of the Czech Republic (Projects No. MSM-0021622419, No. LC535 and No. 2B06052).

References

1 Yuri Boykov and Gareth Funka-Lea. Graph cuts and efficient n-d image segmentation (review). *International Journal of Computer Vision*, 70(2):109–131, 2006.

2 Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, 2004.

3 Yuri Boykov and Olga Veksler. *Handbook of Mathematical Models in Computer Vision*, chapter Graph cuts in vision and graphics: Theories and Applications, pages 79–96. Springer-Verlag, 2006.

4 Tony F. Chan and Luminita A. Vese. Active contours without edges. *IEEE Transactions on Image Processing*, 10(2):266–277, February 2001.

5 L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

6 A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 136–146, 1986.

7 Xiao Han, Chenyang Xu, and Jerry L. Prince. A topology preserving level set method for geometric deformable models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25:755–768, 2003.

8 Reinhard Klette and Azriel Rosenfeld. *Digital Geometry: Geometric Methods for Digital Picture Analysis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

9 Vladimir Kolmogorov and Ramin Zabih. What energy functions can be minimized via graph cuts? *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(2):147–159, 2004.

10 G. Malandain and G. Bertrand. Fast characterization of 3d simple points. In *11th International Conference on Pattern Recognition*, pages 232–235, 1992.

11 Stanley J. Osher and Ronald P. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer, 1 edition, October 2002.

12 S. Vicente, V. Kolmogorov, and C. Rother. Graph cut based image segmentation with connectivity priors. In *CVPR 2008: IEEE Conference on Computer Vision and Pattern Recognition.*, pages 1–8, jun. 2008.

13 Yun Zeng, W. Chen, and Q. Peng. Efficiently solving the piecewise constant mumford-shah model using graph cuts. Technical report, Dept. of Computer Science, Zhejiang University, P.R. China, 2006.

14 Yun Zeng, Dimitris Samaras, Wei Chen, and Qunsheng Peng. Topology cuts: A novel min-cut/max-flow algorithm for topology preserving segmentation in n-d images. *Computer Vision and Image Understanding*, 112(1):81–90, 2008.

Haptic Rendering Based on RBF Approximation from Dynamically Updated Data

Jan Fousek[1], Tomáš Golembiovský[1], Jiří Filipovič[1], and Igor Peterlík[2]

1 **Faculty of Informatics, Masaryk University, Czech Republic**
2 **Institute of Computer Science, Masaryk University, Czech Republic**

Abstract

In this paper, an extension of our previous research focused on haptic rendering based on interpolation from precomputed data is presented. The technique employs the radial-basis function (RBF) interpolation to achieve the accuracy of the force response approximation, however, it assumes that the data used by the interpolation method are generated on-the-fly during the haptic interaction. The issue caused by updating the RBF coefficients during the interaction is analyzed and a force-response smoothing strategy is proposed.

Keywords and phrases haptic rendering, radial-basis function approximation, precomputation, deformation modeling

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.26

1 Introduction and Related Work

The real-time haptic interaction with deformable objects is an interesting area of research with wide range of applications. However, the haptic rendering of response force that requires high refresh rate (at about 1 kHz), heavily restrict the cost of computations needed for updating the force. Nevertheless, the expensive computations must be involved when for example realistic deformation modeling is considered, motivated by construction of surgical simulators for training or operation planning. In this case, the behaviour of the scene is goverened by models emerging in the theory of elasticity, resulting in computationally expensive calculations that cannot be performed inside the haptic loop.

Besides various strategies such as simplification of the underlying mathematical models, an approximation from precalculated data has been successfully exploited. A pioneering work employing the force extrapolation to achieve the haptic refresh rate for interaction with linear anisotropic deformable model is given in [10]. In [6], the force response is computed in haptic loop by linear interpolation of precalculated deflection curves stored in mesh nodes. A data-driven haptic rendering based on the radial-basis function (RBF) interpolation of measured data is studied in [4]. The work is further extended in [5] by considering the interpolation for reproducing also the viscoelastic properties. A comprehensive system based on RBF-based neural networks is described in [1].

In this paper, an extension our earlier work based on precomputation and approximation of state spaces [8] is presented. Originally, the method employed fast polynomial interpolation using regularly distributed data generated during off-line phase. Two modifications of the approach has been proposed: first, in [9], it was shown that the accuracy of the method can be significantly improved by employing radial-basis function (RBF) interpolation. Second, in [3], the original method has been modified to avoid the time-consuming recalculation of state space by introducing on-line precomputation when the states needed for interpolation are generated directly during the haptic interaction. However, it was supposed that only

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 26–31

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

polynomial interpolation can be used for that case. The main contribution of this paper is to show that two modifications described above can be combined resulting in technique taking advantage from both the on-line state generation and RBF approximation.

2 Preliminaries

2.1 Radial-Basis Function Interpolation and Haptic Rendering

The radial-basis function (RBF) interpolation is used for approximating a function $f(x)$ in arbitrary point x from a given vector of evaluations $\mathbf{f} = (f_1, \dots f_n)$ in arbitrarily distributed points $(x_1, \dots x_n)$. It is computed as

$$\tilde{f}(x) = p(x) + \sum_{i=1}^{n} \lambda_i \phi\left(|x - x_i|\right) \tag{1}$$

where ϕ is the radial-basis function, $p = c_0 + c_1 x \dots c_m x^m$ is a polynomial and λ_i are interpolation coefficients. The vectors $\boldsymbol{\lambda}$ and $\mathbf{c}$ of coefficients λ_i and c_i, respectively, are computed from the evaluation vector $\mathbf{f}$ as follows:

$$\begin{pmatrix} \mathbf{A} & \mathbf{P} \\ \mathbf{P}^\top & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{\lambda} \\ \mathbf{c} \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ 0 \end{pmatrix} \tag{2}$$

where $A_{ij} = \phi\left(|x_i - x_j|\right)$ and $P_{ij} = p_j(x_i)$ for basis polynomial p_j. In our setting, linear polynomial p is considered together with linear ($\phi(r) = r$) and cubic ($\phi(r) = r^3$) radial-basis functions.

In [9], the RBF method is employed to approximate the components of the force response $f(x)$ for an arbitrary position x of the haptic interation point (HIP) during haptic loop. However, before the haptic interaction is executed, a large set of force responses f_i associated to various positions x_i of HIP (being referred as *state space*) is constructed numerically during an off-line phase.

2.2 State-space Construction

In [3], *on-line construction* of the state space has been introduced: instead of building the entire state-space during the off-line phase, the calculation of a force responses is performed directly during the interaction. The distribution of the points x_i of the state space for which the responses f_i are being calculated is determined on-the-fly by the actual position and motion of the HIP.

It has been shown that although the precomputation of a single state (i.e. the force response) takes from hundreds milliseconds to seconds, the interpolation runs stably at haptic refresh rate (1 kHz), provided the states are generated concurrently by dozens of processes running in distributed environment. Thus, the state space used by interpolation is gradually augmented: in our setting from about dozens to hundreds of new states were added to the actual space during one second. It was supposed that in each step of the haptic loop the HIP is located within the area enclosed by regularly-distributed known states, so the polynomial interpolation can be used to approximate the associated force response. However, due to this assumption, the HIP speed is restricted as quantitatively evaluated in [3].

3 RBF Approximation on Updated Data

3.1 From Interpolation towards Extrapolation

The issue addressed in this paper is represented by the limited speed of motion towards the horizon of the actually known state space. In contrast to the polynomial interpolation using the regularly distributed data, the approximation based on RBF is able to make use of data scattered in the space. Therefore, it is possible to leave the space fully populated by already known states and approximate the values of the force response in areas with sparsely distributed known states and even extrapolate outside the known state space. The downside of this approach is increase of the approximation error and possible loss of the accuracy.

Further, in contrary to the interpolation based on regular grid, the RBF method can make use of arbitrary number of known points. Therefore, two tasks arise to be performed by the process handling the interpolation :

1. select up to n nearest points, where n is constant and
2. compute the RBF coefficients for selected points by solving Eq. 2.

There is no need to explicitly switch between extrapolation and interpolation, the RBF method approximates the forces in the same way both outside and inside the area defined by the known states. When leaving the densely precomputed area and the approximation shifts to the extrapolation, numerous distributions of the nearest known points can occur. Following distributions describe the characteristic situations with respect to the HIP movement which can occur during the simulation (see figure 1 for illustration):

- HIP enters a **sparse area** where a limited number of states is available: RBF starts partially extrapolating from the remote points in the safe area,
- HIP gets into **gap**: RBF provides a bridge between two safe areas,
- HIP gets into **blank area** where no states are known: RBF works as pure extrapolation method.

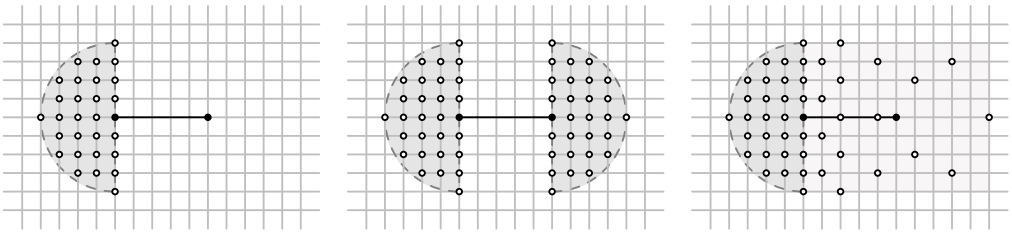


Figure 1 Characteristic situations when leaving the densely-populated area. The known states are shown on the regular grid together with the path of HIP. From left to right: sparse area, bridge and blank area.

3.2 Returning from Extrapolation towards Interpolation

The distribution of known states with respect to the position of HIP is updated for two reasons: either new states requested previously arrive from the solvers, or the position of the HIP changes rendering some new known states as nearer than some previously used states. In both cases, the blank area can change into sparse area or gap and further into safe area shifting the approximation back from extrapolation to interpolation.

To reflect the changes of the state space inside the approximation procedure, it is necessary to compute updated set of RBF coefficients. However, two issues arise here: first, the complexity of solving Eq. 2 is in $O(n^3)$ and therefore for a large number of states, recomputing the coefficients can be a computationally demanding task that cannot be done inside the haptic loop. Second, switching to a new set of RBF coefficients in one step can result in non-continuous change in the force response being perceived as a jump in the force feedback. Therefore the interpolation is implemented as two concurrent processes:

Updating process is event-driven: if a state space is updated or HIP travels a distance larger than predefined threshold, it selects n known states enclosing the HIP and computes the updated RBF coefficients $(\boldsymbol{\lambda}', \mathbf{c}')$ according to Eq. 2.

Interpolating process is running at haptic rate: in each step, it computes the force response from the actual RBF coefficients $(\boldsymbol{\lambda}, \mathbf{c})$ according to Eq. 1. If updated set of coefficients $(\boldsymbol{\lambda}', \mathbf{c}')$ is available, it performs a *damped coefficient switch*, i. e. in following steps of the haptic loop it computes interpolated forces from both $(\boldsymbol{\lambda}, \mathbf{c})$ and $(\boldsymbol{\lambda}', \mathbf{c}')$ and calculates weighted average moving the weight in every tenth step (10 ms) between $(\boldsymbol{\lambda}, \mathbf{c})$ and $(\boldsymbol{\lambda}', \mathbf{c}')$ so that the force difference is lower than 7% which is a lower bound for the just noticeable difference [7, 11] for variety of haptical hand interactions.

By shifting the weight between two successive RBF coefficients on lower frequency than is the human perceptive resolution and by limiting the change of resulting force by the just noticeable difference we can assure the user will not experience twitches or vibrations caused by discontinuous changes in the force feedback.

4 Results

We have incorporated the modifications presented in previous section in our haptical simulator [2] and present here the evaluation of this implementation. Also to be able to study the effects of dynamical changes in the set of precomputed set of states and to simulate the on-line precomputation in deterministic way, we have modified the scheduler so that it was able to add the states to the known set with in advance given timing.

All the experiments have been performed on PC with 2x Dual Core AMD Opteron Processor 2218 2.6 GHz with 3 GB RAM using the model of the human liver with 1777 elements (see [9] for details). The total length of model was about 22 cm and we have experimented with force response associated with compression-like deformation between 0 and 6 cm. The state-space was given by a regular grid with cell size of 6 mm.

First, the experiments have confirmed that it is possible to compute the approximation using the RBF inside the haptic loop, whereas the RBF update spreads over dozens of iterations of the haptic loop. For example, for 100 states it takes in average 160 ms to update the RBF and 60 µs to approximate the forces, for 300 states it is 1.6 s and 200-250 µs, respectively.

Further, the magnitude of extrapolation error has been studied (Fig. 2). These results were obtained by displacing one node of the mesh by 6 cm in the direction of the x axis towards the center of the model considering three characteristic situations introduced in section 3.1. Note that presented values represent time independent equilibrium and is therefore not influenced by the damping effect. The absolute and relative errors are given w.r.t. the completely known state space. In all the cases state space of 100 nearest known states was used. In the first case the displacement path started at the border of the known space and continued towards space without any precomputed values. In the second case, it crossed the empty space towards another area continuously populated with known states.

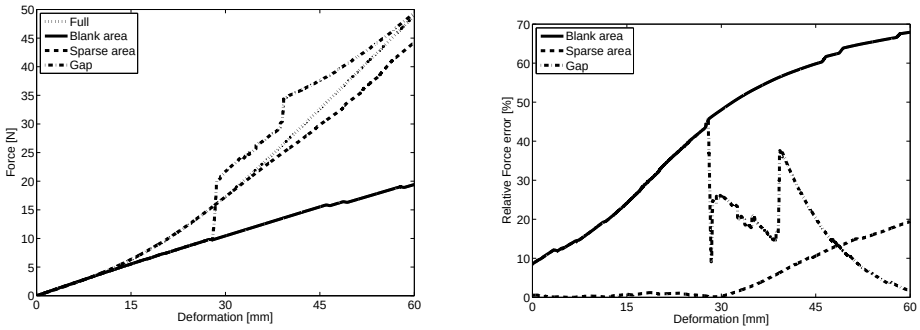


Figure 2 Resulting force (left) and relative error (right) of the extrapolation for the three characteristic situations from section 3.1.

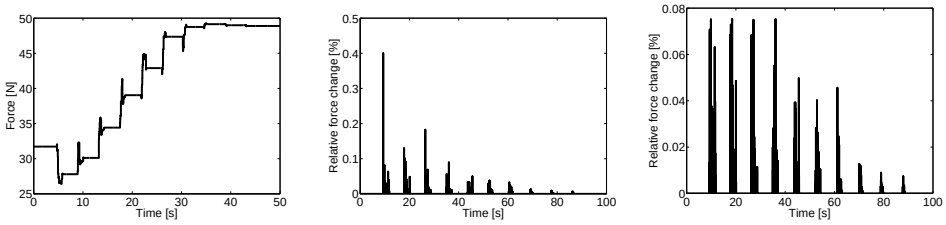


Figure 3 Damping the RBF coefficient switch: comparison of the force response change after comming sets of states. Absolute force (right) is presented together with ratio between resulting force for sucessive haptic frames an non-damped (middle) and damped (right) coefficient switching.

In the case of sparse area, the density of known states descended continuously along the deformation path.

The error of the extrapolation in the case of blank area grew rapidly with the length of deformation while few sparsely distributed known states along the path were enough for the RBF to approximate the resulting force with relatively low error. Also note two rapid changes in the approximated force in the case of the gap crossing. These come from the change in the set of nearest points used for the approximation: the first jump corresponds to the point where first points from the other side of the gap are added, the second jump then similarly corresponds to the removing of the last points from the starting side of the gap.

Finally, the damped switch of the RBF coefficients was verified experimentally. The HIP was displaced similarly to the blank area setup in the area without any precomputed states extrapolating from distant continuously precomputed area. Then in intervals of 10 s groups of states were added to the known state space starting with most distant states to the HIP filling the empty space completely in the end. The Fig. 3 depicts jumps in the absolute value of feedback force together with relative change of the force between two successive iterations of the haptic loop and it can be clearly seen, that without damping the change exceeds the just noticeable difference by far, whereas the proposed damping was able to spread the jump in the force in several successive frames of the haptic feedback.

5 Conclusions and Future Work

In this paper, an extension to the haptic rendering method based on precomputation and approximation has been proposed. It was shown that the RBF interpolation, which improves the accuracy of the approximation, can be employed also in the case when the states are generated directly during the haptic interaction. Possible issues were identified and resolved: the RBF coefficients were recomputed by additional thread running on lower frequency and the switch to the updated coefficients were damped in order to avoid artificial discontinuities in the force response. The technique proposed in the paper was verified by experiments presented in result section.

Our future work will focus on finding new scheduling strategies considering e. g. motion direction heuristics, sparse area precomputation or prioritizing and preempting some states. We also plan to improve the speed of RBF coefficient update by using some advanced technique for matrix inversions.

Acknowledgements The financial support provided by Ministry of Education, Youth and Sport of the Czech Republic under the research program MSM0021622419 is acknowledged. The access to the METACentrum computing facilities provided under the research intent MSM6383917201 is acknowledged.

References

1 D. Deo and S. De. Phyness: A physics-driven neural networks-based surgery simulation system with force feedback. In *Proc. of 3rd Joint Eurohaptics Conference and Haptic Symposium*, pages 30–34, 2009.

2 J. Filipovič, I. Peterlík, and M. Madzin. Parallel real-time deformation simulator. `http://parides.sourceforge.net/`, 2010.

3 J. Filipovič, I. Peterlík, and L. Matyska. On-line precomputation algorithm for real-time haptic interaction with non-linear deformable bodies. In *Proc. of The 3rd Joint EuroHaptics Conf. and Haptic Symposium*, pages 24–29, 2009.

4 R. Hover, M. Harders, and G. Szekely. Data-driven haptic rendering of visco-elastic effects. In *Proc. of Symp. on Haptic Interf. for Virt. Env. and Teleop. Systems*, pages 201–208, 2008.

5 R. Höver, G. Kósa, G. Székely, and M. Harders. Data-driven haptic rendering–from viscous fluids to visco-elastic solids. *IEEE Trans. Haptics*, 2(1):15–27, 2009.

6 M. Mahvash and V. Hayward. Haptic simulation of a tool in contact with a nonlinear deformable body. In *Proc. of Surg. Sim. and Soft Tissue Def.*, pages 311–320, 2003.

7 X.D. Pang, H.Z. Tan, and N.I. Durlach. Manual discrimination of force using active finger motion. *Perception & Psychophysics*, 49(6):531–540, 1991.

8 I. Peterlik and L. Matyska. An algorithm of state-space precomputation allowing non-linear haptic deformation modelling using finite element method. In *Proc. of the 2nd Joint EuroHaptics Conference and Haptic Symposium*, pages 231–236, 2007.

9 I. Peterlík, M. Sedef, C. Basdogan, and L. Matyska. Real-time visio-haptic interaction with static soft tissue models having geometric and material nonlinearity. *J. of Comp.& Graph.*, 34(1):43 – 54, 2010.

10 G. Picinbono, J-C. Lombardo, H. Delingette, and N. Ayache. Improving realism of a surgery simulator: linear anisotropic elasticity, complex interactions and force extrapolation. *Jour. of Vis. and Comp. Anim.*, 13(3):147–167, july 2002.

11 H.Z. Tan, X.D. Pang, and N.I. Durlach. Manual resolution of length, force, and compliance. *Advances in Robotics*, 42:13–18, 1992.

Modeling Gene Networks using Fuzzy Logic

Artur Gintrowski

Silesian University of Technology, Institute of Electronics,
Division of Microelectronics and Biotechnology
16 Akademicka Street, 44-100 Gliwice, Poland
artur.gintrowski@gmail.com

Abstract

Recently, almost uncontrolled technological progress allows so called high-throughput data collection for sophisticated and complex experimental biological systems analysis. Especially, it concerns the whole cellular genome. Therefore it becomes more and more vital to suggest and elaborate gene network models, which can be used for more complete interpretation of large and complex data sets. The presented paper concerns modeling of interactions in yeast genome. With the reference to previously published papers concerning the same subject, our paper presents a significant improvement in calculation procedure leading to very effective reduction of time of calculation.

1998 ACM Subject Classification I.5.1 Models, I.5.4 Applications

Keywords and phrases Fuzzy network, gene expression, time optimization

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.32

1 Introduction

Immediate technological evolution allows the analysis of more and more composite biological systems. The creation of elaborate gene network models involves widely developed analyses, which allows the better utilization of biological interpretation of medical data packages, particularly data concerning gene expression measurements. An example of a very effective modeling of gene network was previously presented by Sokhansanj et al. in BMC Bioinformatics [1]. This algorithm, which takes advantage of the theory of fuzzy sets, allows the creation of a model of intergenetic interactions. Input data for the described fuzzy system are gene expression measurements obtained as a result of a biological experiment using GeneChip microarray technology.

The huge advantage of the described method is that it receives the exact model of interactions in the result of the analysis. However, the time of account is its main defect. In the article, we present a detailed description of the algorithm with modifications on the significant correction of the speed of calculation (over 95% time reduction) to obtain almost identical results in comparison with the original exhaustive algorithm.

2 Original Algorithm Idea

In this study, measurement data of gene expression, obtained during the whole cellular cycle, are used. Results are collected in the I matrix, in which the following rows expressions

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 32–39

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of individual G_i genes are included:

$$I = \begin{bmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,N} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,N} \\ \vdots & & \ddots & \\ e_{M,1} & e_{M,2} & \cdots & e_{M,N} \end{bmatrix} = \begin{bmatrix} G_1 \\ G_2 \\ \vdots \\ G_M \end{bmatrix} \tag{1}$$

$$G_i = [e_1, e_2, \ldots, e_N] \tag{2}$$

2.1 Data Preparation

Raw data are processed non-linearly at the beginning according to (Eq. 3). This transformation conducts the standardization of the input data in the interval $< -1; 1 >$, as it is possible that data are collected from different microarray experiments.

$$\widehat{I} = \frac{\arctan(I)}{\frac{\pi}{2}} \tag{3}$$

Data prepared this way are entered into the fuzzy system, which initially affects their fuzzification for sets with low, medium, and high expression respectively (Eq. 4,5 and 6). In the destination of further calculations, fuzzy data arrays F_L, F_M and F_H are concatenated in the third dimension of the F matrix (Eq. 7).

$$F_{L_{i,j}} = \begin{cases} -\widehat{e}_{i,j} & \widehat{e}_{i,j} < 0 \\ 0 & \widehat{e}_{i,j} > 0 \end{cases} \tag{4}$$

$$F_{M_{i,j}} = 1 - |\widehat{e}_{i,j}| \tag{5}$$

$$F_{H_{i,j}} = \begin{cases} 0 & \widehat{e}_{i,j} < 0 \\ \widehat{e}_{i,j} & \widehat{e}_{i,j} > 0 \end{cases} \tag{6}$$

$$F = \{F_L, F_M, F_H\} \tag{7}$$

2.2 Fuzzy Rules

The database in the described fuzzy system consists of three basic fuzzy rules in the following form:

- if *input* is ***LOW*** then *output* is ***ExpressionLevel***
- if *input* is ***MEDIUM*** then *output* is ***ExpressionLevel***
- if *input* is ***HIGH*** then *output* is ***ExpressionLevel***

where: *ExpressionLevel* $\in$ {*LOW, MEDIUM, HIGH*}.

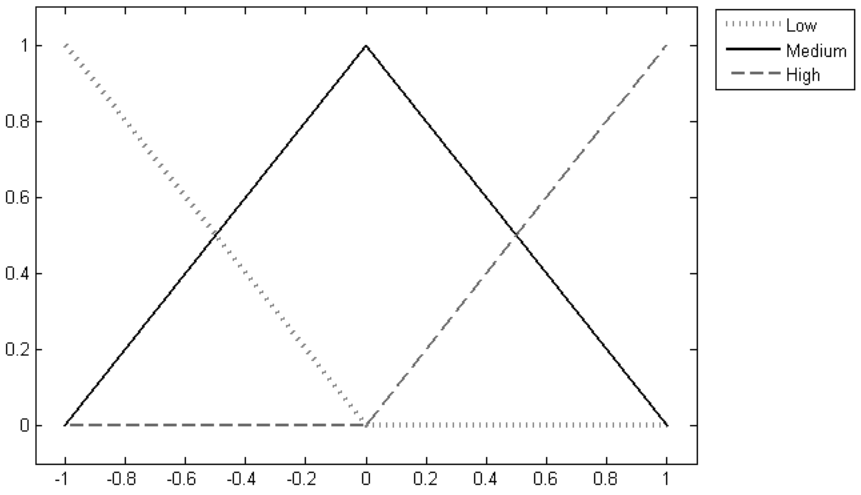


Figure 1 Linear functions used to fuzzify gene expression data

With a view to enable calculations, a relevant notation of the record is provided in the form of the r vector:

$$r = [l_1, l_2, l_3] \tag{8}$$

where l_1 is the output expression if input is *LOW*, l_2 is the output expression if input is *MEDIUM*, l_3 is the output expression if input is *HIGH* and $l_i \in \{1, 2, 3\}$ (output expression is 1 – low, 2 – medium or 3 – high).

Example: The following exemplary rule database

- if *input* is *LOW* then *output* is *HIGH*
- if *input* is *MEDIUM* then *output* is *MEDIUM*
- if *input* is *HIGH* then *output* is *LOW*

can be described using the following vector:

$$r_{example} = [3, 2, 1]$$

2.3 Iteration Issue

The fuzzy system is used repeatedly to model the initial vector of the expression, based on the chosen input genes and the chosen linguistic fuzzy rules.

For every choice of initial genes and combination of linguistic rules (combination of vectors r), the result of a vector being compared with the original vector of the expression of the initial gene is obtained. The optimum combination of input genes and linguistic rules is chosen using a certain rate of error described hereinafter. This guarantees gaining the highest resemblance

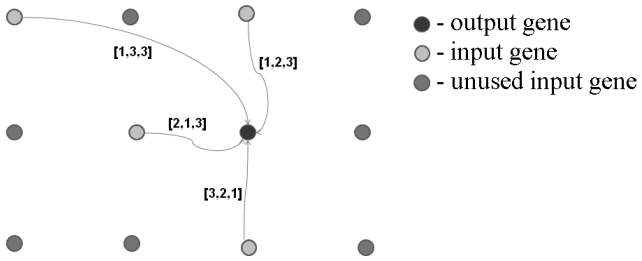


Figure 2 Example of the connections to an output gene in the fragment of the network modeled using four input genes per output

between the vector obtained as a result of the modeling and the original vector of the initial gene created.

Due to the huge amount of iterations, limiting the number of genes considered in the analysis to be initial genes is intentional. An optimum number of four initial genes is suggested.

Determining the huge number of iterations discarded is worthwhile. In the analysis of the discussed microarray of the 12 genes of yeast, conducting the analysis of the effect of the 11 remaining genes is necessary for each of them. To establish the recommended number of four entries, conducting calculations for the following number of the combinations of input genes is necessary:

- C_{11}^1 combinations for tests of influence of one gene on the network output
- C_{11}^2 combinations for two inputs
- C_{11}^3 combinations for three inputs
- C_{11}^4 combinations for four inputs.

These values are calculated in (Tab. 1). In the case of testing the influence of one gene, for each output the test of the 27 combinations of linguistic rules is necessary. For two input genes, there are $27^2 = 729$ combinations of rules; and for three input genes, there are $27^3 = 19683$ combinations. The case with four input genes gives $27^4 = 531441$ combinations. The total number of iterations included in the elaborated software during the single yeast microarray analysis gives

$$L_{oryg} = 12 \cdot C_{11}^1 \cdot 27 + C_{11}^2 \cdot 27^2 + C_{11}^3 \cdot 27^3 + C_{11}^4 \cdot 27^4 = 2143963404.$$

They require compound accounts from the disposed implementation programme on time optimization executable for over two billion cases of the described procedures. The final modification of the conclusion process contributes to the progress through the introduction of additional ratios of error in the analysis of the combinations of input genes. This most significantly limits the number of selected fuzzy rules.

2.4 Inference Engine

The described fuzzy system concludes each iteration of the algorithm, that is, for each combination of the fuzzy input genes F_C (see Eq. 7) as well as the combination of linguistic

Table 1 Number of analyzed combinations of rules for each combination of input genes

$C_{11}^1 = \binom{11}{1} = \frac{11!}{(11-1)!\cdot 1!} = \frac{11!}{10!} = 11$
$C_{11}^2 = \binom{11}{2} = \frac{11!}{(11-2)!\cdot 2!} = \frac{11!}{9!\cdot 2!} = 55$
$C_{11}^3 = \binom{11}{3} = \frac{11!}{(11-3)!\cdot 3!} = \frac{11!}{8!\cdot 3!} = 165$
$C_{11}^4 = \binom{11}{4} = \frac{11!}{(11-4)!\cdot 4!} = \frac{11!}{7!\cdot 4!} = 330$

rules included in the matrix R_C (Eq. 8).

$$R_C = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix} \tag{9}$$

$$\widetilde{F_{i,j,k}} = F_{C_{i,j,R_{C_{i,k}}}} \tag{10}$$

$$D_{1,j,k} = \sum_{i=1}^{n} \widetilde{F_{i,j,k}} \tag{11}$$

$$D = \{D_{low}, D_{medium}, D_{high}\} \tag{12}$$

As a result, the fuzzy output set D is created.

2.5 Dealing with the Fuzzified Output

To obtain ultimate modeling results, the fuzzy result of inference is transmitted for defuzzification according to equations (Eq. 13 and 15). Equation (Eq. 14) is the graphic interpretation of the conclusion mechanism presented in (Fig. 3).

$$\tilde{O}_{1,i} = \frac{D_{1,i,3} - D_{1,i,1}}{D_{1,i,1} + D_{1,i,2} + D_{1,i,3}} \tag{13}$$

$$\tilde{O} = \frac{D_{high} - D_{low}}{D_{low} + D_{medium} + D_{high}} \tag{14}$$

$$O = \tan(\tilde{O} \cdot \frac{\pi}{2}) \tag{15}$$

The expression model O obtained through the presented algorithm on the output gene is compared with the original expression vector of the output gene G_O according to the following formula:

$$E = \frac{\sum_{i=1}^{N} (G_{O_i} - O_i)^2}{\sum_{i=1}^{N} (G_{O_i} - \overline{G_O})^2} \tag{16}$$

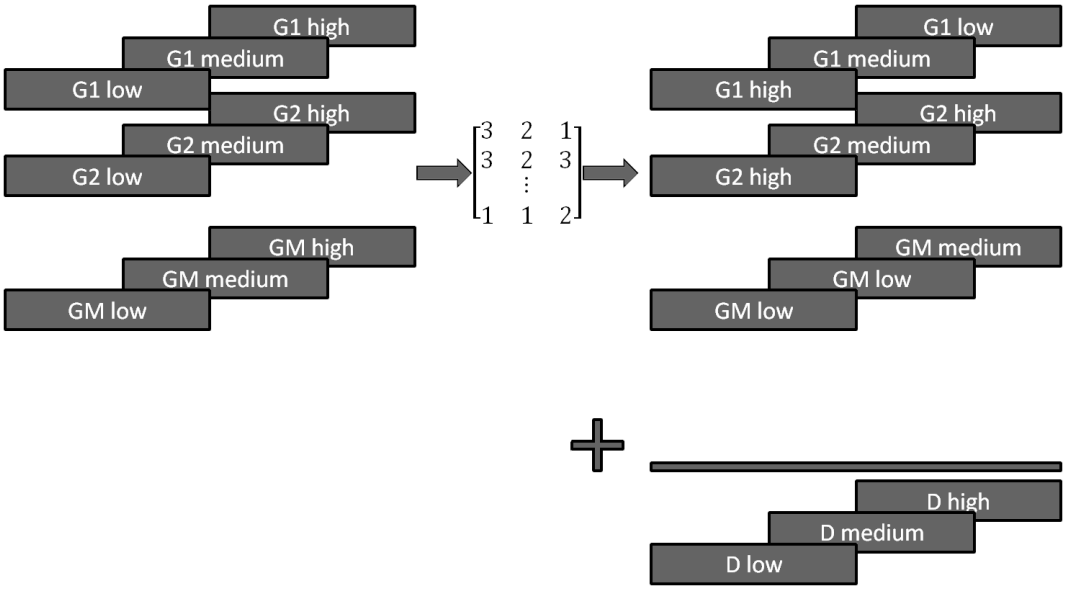


Figure 3 Inference scheme using the exemplary rule matrix. Inputs – genes $G_1, G_2, \ldots, G_M$. Fuzzified output – D matrix.

where G_{O_i} is the i–th expression measurement of the original output gene, O_i is the i–th expression vector measurement obtained as a modeling result and $\overline{G_O}$ represents the mean of the expression vector obtained as a result of modeling. In the comparison between the obtained identical model and the real expression vector of the output gene, the error coefficient E takes a value of zero. Hence, the better the choice of input genes and their respective linguistic rules, the lower the value of error coefficient is.

3 Time Optimization

As shown in (Sec. 2.3), computational complexity of the algorithm results mainly from the necessity for the continuous repetition of the fuzzy conclusion procedure for the huge number of combinations of the applied input data. To reduce that amount in the conclusion mechanism, several modifications are applied using three additional error coefficients constructed analogously to the main error coefficient (Eq. 16). However, they work inside the fuzzy system and on the fuzzified data of input genes, as well as in the fuzzy interpretation of the original output gene.

$$E_L = \frac{\sum_{i=1}^{N}(F_{L_{O_i}} - D_{low})^2}{\sum_{i=1}^{N}(F_{L_{O_i}} - \overline{F_{L_O}})^2} \tag{17}$$

$$E_M = \frac{\sum_{i=1}^{N}(F_{M_{O_i}} - D_{medium})^2}{\sum_{i=1}^{N}(F_{M_{O_i}} - \overline{F_{M_O}})^2} \tag{18}$$

$$E_H = \frac{\sum_{i=1}^{N}(F_{H_{O_i}} - D_{high})^2}{\sum_{i=1}^{N}(F_{H_{O_i}} - \overline{F_{H_O}})^2} \tag{19}$$

The modified algorithm has four steps. In the first three, the k best linguistic rules with respect to the smallest coefficients E_L, E_M and E_H are stored in particular fuzzy sets. Next, in the fourth step, the analysis of the original algorithm is subsequently performed, taking only k^3 of all the best combinations of the linguistic rule vectors stored in the first three steps.

For comparison, the number of fuzzy conclusion procedure calls in case of the optimized algorithm for the described yeast microarray is equal to

$$L_{opt} = 12 \cdot C_{11}^1 \cdot (3 \cdot 3 + k^3) + C_{11}^2 \cdot (3 \cdot 3^2 + k^3) + C_{11}^3 \cdot (3 \cdot 3^3 + k^3) + C_{11}^4 \cdot (3 \cdot 3^4 + k^3).$$

The number for $k = 25$ takes the value of

$$L_{opt}\Big|_{k=25} = 106329168.$$

Therefore, the reduction of the fuzzy conclusion mechanism calls is:

$$(1 - \frac{L_{opt}}{L_{oryg}}) \cdot 100\% = 95.04\%$$

The same percentage of time reduction is also observed.

4 Results

The introduced modifications allow for calculations in a much shorter time. Table (Tab. 2) presents the comparison of the calculation times between the original algorithm and the modified one depending on the constant k of the best rules in the fuzzy sets. As can be seen in the case of the 15 rules, it is possible to obtain the first solution for more than half of the genes using only 1.12% of the original calculation time. For 25 best rules, the calculation time is reduced to 4.9% of the exhaustive search time. Thus, we can obtain the best results for three-fourths of the analyzed genes.

Table 2 Results for the time optimized algorithm

Gene	5 best rules k = 5 time: 0.1%	10 best rules k = 10 time: 0.39%	15 best rules k = 15 time: 1.12%	20 best rules k = 20 time: 2.55%	25 best rules k = 25 time: 4.9%
SIC1	1	1	1	1	1
CLN1	24	11	1	1	1
CLN2	2	2	1	1	1
CLN3	236	18	18	18	5
SWI4	3359	1047	12	12	10
SWI6	293	23	1	1	1
CLB5	121	121	121	121	121
CLB6	7	7	2	1	1
CDC6	1	1	1	1	1
CDC20	4	4	1	1	1
CDC28	58579	12313	722	246	49
MBI1	14	1	1	1	1

References

1 Sokhansanj B., Fitch J., Quong J., Quong A. Linear Fuzzy Gene Network Models Obtained from Microarray Data by Exhaustive Search 5:108 doi: 10.1186/1471-2105-5-108 BMC Bioinformatics, 2004,

2 Fitch J., Sokhansanj B. Genomic Engineering: Moving Beyond DNA Sequence to Function Proc IEEE 88:1949-1971, 2000

Compression of Vector Field Changing in Time*

Tomáš Golembiovský[1] and Aleš Křenek[2]

1 **Faculty of Informatics**
Masaryk University, Czech Republic
nyoxi@ics.muni.cz
2 **Institute of Computer Science**
Masaryk University, Czech Republic
ljocha@ics.muni.cz

Abstract

One of the problems connected with a real-time protein-ligand docking simulation is the need to store series of precomputed electrostatic force fields of a molecule changing in time. A single frame of the force field is a 3D array of floating point vectors and it constitutes approximately 180 MB of data. Therefore requirements on storage grow rapidly if several hundreds of such frames need to be stored.

We propose a lossy compression method of such force field, based on audio and video coding, and we evaluate its properties and performance.

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.40

1 Introduction

Proteins play an essential role in many biological processes in cells and thus are an important subject of studies of many biochemists. Biological activity of a protein is frequently described in terms of the docking problem — under what conditions and where on the surface of the protein a small molecule (ligand) can be attached in an energetically favoured position. Various automated methods of solving the problem exist [3]. Recently haptic-assisted methods (the inter-molecular forces are delivered to the user with a force-feedback device) were introduced [6].

We have developed an improved docking simulator [5] with emphasis on realistic feeling as well as accuracy of the interaction. Unlike previous methods, the ligand is connected to the haptic device by visco-elastic link which is known to improve the feeling of manipulation with real objects [7].

The principal forces involved in the interaction are van der Waals and electrostatic. The latter have long range influence, therefore the only feasible approach is precomputing the resulting force field on a 3D grid of sufficient resolution (typically 250 points in each dimension). Because the molecule can change its shape in time, the induced force field changes as well, resulting in a long sequence of 3D "frames" whose size becomes unmanageable. Here we propose a compression method on this data inspired by common audio and video compression.

2 Compressed Data and Its Properties

The data we are interested in are formed by a series of frames. A single frame is a 3D array of 3D vectors, having 250 cells in each dimension. The force vectors are represented as single

* This work was done as a part of the research intent MSM0021622419. The access to the MetaCentrum supercomputing facilities provided under the research intent MSM6383917201 is highly appreciated.

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 40–46

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

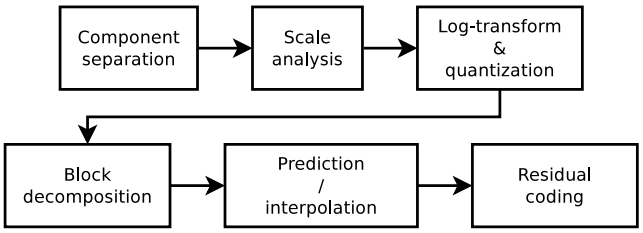


Figure 1 Schema of lossy compression of force field

precision floating point numbers. Altogether it leads to approx. 180 MB ($250^3 \cdot 3 \cdot 4$B) of raw data per one frame.

The force field is a sum of contributions of all charged atoms of the macromolecule on a unit charge, given by the Coulomb law $F_i = k_e q_i e / r^2$. This yields steep peeks in near vicinity of the macromolecule atoms while the field is relatively smooth farther. The dynamic range is as many as 18 orders of magnitude for the same reasons. On the other hand, due to the complementary van der Waals interaction in the haptic model, which generates even steeper barriers, it is virtually impossible for any two atoms to get closer than approx. 1 Å. Therefore we do not have to insist on accurate encoding of the force field peeks.

3 Compression Method

According to the reasoning given above, accurate encoding of high values of the force field is not required, therefore we trade off accuracy for compression ratio, and we aim at a rather aggressive lossy compression scheme.

Currently, the Cartesian components of force vectors in each frame are processed separately, exploiting their possible correlation would be a subject to further work.

Figure 1 outlines the whole compression process. Its principal steps, heavily inspired by audio and video codecs, are:

1. Component separation: Cartesian components are treated independently.
2. Scale analysis: for each frame set an appropriate scaling factor (which maps the field domain onto $[-1; 1]$) for subsequent μ-law transform is determined.
3. Logarithmic transform and quantization
4. Decomposition into blocks
5. Prediction: best-fitting parameters of some prediction model are found
6. Coding of residuals w.r.t. the predictor

The rest of this section deals with detailed description of those steps.

3.1 Frame Types

The frames of the force field are classified in a manner defined in MPEG-1 and later adopted in many video codecs.:

1. I-frame — uses linear filters with fixed coefficients
2. P-frame — uses one previous I- or P-frame for prediction
3. B-frame — uses one previous and one following I/P-frame for prediction

Specific assignment of frame type is controlled by configuration. One example of such a scheme is `IBBPBBP` that is commonly used by MPEG video compression.

3.2 Logarithmic Transform and Quantization

The first approach of compressing the floating point numbers directly lead to low compression rates and high decompression times due to large amount of floating point arithmetics. Therefor quantization was introduced combined with logarithmic transform to deal with the large dynamic range of the data (Sect. 2). Values are scaled down in order to fit into the interval [-1;1]. Then continuous version of μ-law companding algorithm is applied:

$$F(x) = \operatorname{sgn}(x)\frac{\ln(1+\mu|x|)}{\ln(1+\mu)}$$

We allow adaptive scaling to accommodate different value range. However, since I- and P-frames are used for prediction later, the scale cannot change arbitrarily. Therefore we split frames into sets by the location of I-frames and let all frames in the set share one scale.

The transformed values are subsequently quantized — expressed as integer value with configurable number of bits. Finally, the whole field is split into smaller blocks of equal size that are further processed separately.

3.3 Prediction

3.3.1 I-frames

I-frames start each frame set and they represent the corresponding frame accurately (up to the loss due to logarithmic quantization). This is to allow seeking in the compressed file and to prevent accumulation of further error throughout the whole frame sequence.

In I-frames the values (each Cartesian component of the force) in the block (cube of points) are ordered into a sequence according to a fixed pattern first, and those are passed to predictor. The force fields are formed by continuous wave-like patterns that makes them in some aspects similar to audio waves. For prediction we use linear order predictors with fixed coefficients, originally described in the Shorten[9], where they were used for compression of audio signal, and were later adopted also by AudioPaK[2] or Flac[1]. The predictor attempts to fit a p-order polynomial to the last p points:

$$\hat{x}_0[n] = 0 \tag{1}$$
$$\hat{x}_1[n] = x[n-1] \tag{2}$$
$$\hat{x}_2[n] = 2x[n-1] - x[n-2] \tag{3}$$
$$\hat{x}_3[n] = 3x[n-1] - 3x[n-2] + x[n-3] \tag{4}$$
$$\dots \tag{5}$$

where $x[n]$ denotes signal sample at point n and $\hat{x}_k[n]$ the k^{th} order prediction for the sample at point n. The prediction error can be also computed recursively, making the method attractive from the performance point of view. We chose 5 as the maximal order used. Higher order predictors showed only little contribution to the compression while slowing down the compression significantly. The reader is suggested to refer to [9] for further details.

3.3.2 P-frames

P-frames use previous I- or P- frame as the prediction that is then subtracted from current frame to obtain the prediction residual (encoded in the same way as I-frames, cf. Sect. 3.4). Unlike MPEG compression we don't make motion prediction yet; so far we were not able to elaborate a promising scheme.

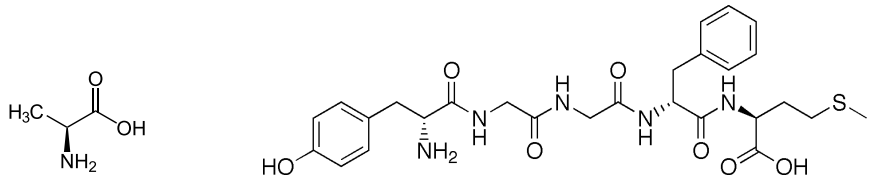


Figure 2 Testing molecules alanine and enkephalin

3.3.3 B-frames

B-frames use the nearest preceding and following I- or P-frame from which weighted average is computed. The weights used are distances (number of frames) from current frame to the respective I- or P-frame:

$$\hat{X} = \frac{\triangle f \cdot P + \triangle p \cdot F}{\triangle p + \triangle f}$$

where P and F are the preceding and following frame, and $\triangle p$ and $\triangle f$ are the distances from them. The same residual encoding (Sect. 3.4) is used again.

3.4 Residual Coding

The distribution of residual values resembles a two-sided geometric distribution. Therefore we choose Rice codes [8], which are known to perform well in this case, for the residual coding.

Rice codes are designed to code non-negative integer numbers, we have to map the error codes to this domain. If all error values in a block are either non-positive or non-negative (which turns to be more than 50% of cases on real data), we encode just absolute values and keep the block-wide information on the sign. This trick saves one bit per single value. Otherwise the values are interleaved — positive values map to even numbers and negative values to odd numbers.

3.5 Frame Promotion

It turns out that for some B-frames the P-frame prediction would lead to a better compression. Moreover, evaluation of the experimental data shows that it typically happens on all Cartesian components of the vector field simultaneously. Therefore we introduce *frame promotion*, ad-hoc replacement of the B-frame by P-frame in the specific frame set.

The method requires no extra work on the decompression side and only small additional effort during the compression. The overall compression ratio is improved by 0.5%–2% (see Sect. 4.2).

Similarly, sometimes I-frames deliver better compression than P-frames as well. However, we do not promote P- to I-frames due to much higher requirements on decompression time.

4 Results

Experimental implementation of the presented compression method is available. We used it for quantitative analysis of the method properties. Rather than evaluating the method directly with large macromolecules, we chose smaller molecules — alanine amino-acid and met-enkephalin peptide (Fig. 2). Alanine itself and the amino-acids forming enkephalin occur frequently in large macromolecules, and due to their flexibility (there are two freely rotatable bonds in alanine, and many more in enkephalin) they represent the worst case of

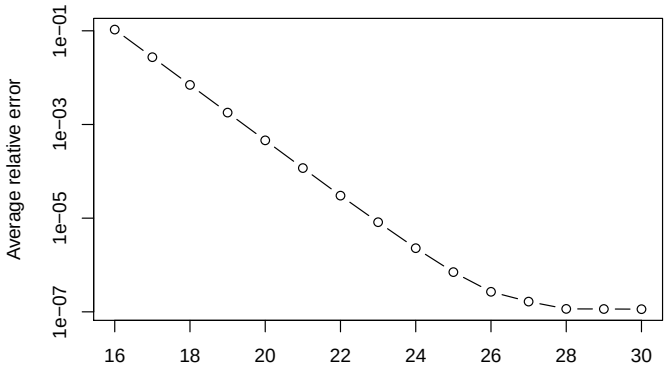


Figure 3 Average relative error for enkephalin at various quantization levels

type	alanine	enkephalin
I	17%	16%
P	6%	6%
B	7%	7%

Table 1 Compression ratio per frame type. Only P-frames of compression schemes without B-frames are included (see text)

local behaviour of the force field for the compression. Testing data sets were generated by molecular dynamics simulation, interpolated with a method we developed before [4]. The force field was sampled on a grid of 250 points in each dimension, a resolution used in the haptic application, with approx. 200 frames in both sets.

4.1 Data Loss

Figure 3 shows typical relative loss observed. Data loss is only a result of floating point arithmetic involved in μ-law transform and the quantization. Tests were performed with quantization ranging from 16 to 30 bits. Under the assumption that the compression inaccuracies are located close to the macromolecule atoms (Sect. 2) we conclude that the relative error of 10 % at 16 bits is still acceptable.

4.2 Compression Ratio

Table 1 shows the achieved compression ratio per frame type and testing data set, using 16bit quantization. The dispersion of the compression ratio for I- and B-frames is negligible regardless of their position in the compression scheme. On the other hand, for P-frames it holds only when they follow an I- or P-frame immediately in the scheme. This is a consequence of the prediction used for P-frames — the farther a P-frame is from the preceding I- or P-frame, the bigger is the prediction error, making the space occupied by encoded residuals to grow almost arbitrarily.

We didn't observe any significant difference between behaviours of the compression of the two molecules, therefore we further refer to enkephalin only.

Figure 4 shows results for several compression schemes applied on the enkephalin data set. The testing frame sequence was split fruther into four phases — two are fairly stable,

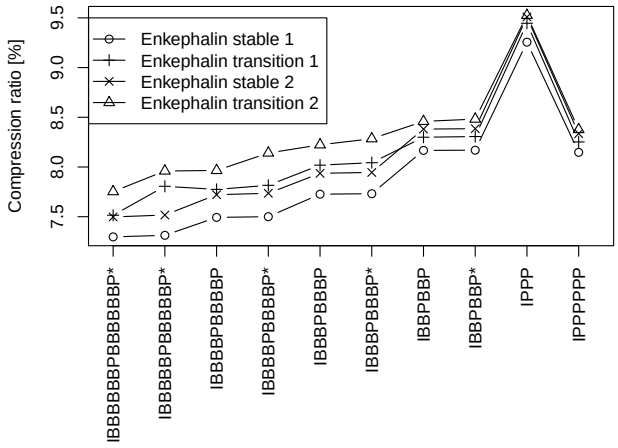


Figure 4 Comparison of compression schemes Asterisk marks disabled frame promotion.

where the shape changes are small in either unfolded or folded shape of the molecule, while two others represent folding and unfolding of the molecule. The absolute compression ratios differ between these segments, however, the trends are quite parallel.

We tested two schemes without B-frames and several with two groups of B-frames (the *IBBPBBP* type), each group having two to six B-frames. The trend visible in the figure indicates that adding groups of more than three B-frames improves the overall compression ratio w.r.t. P-frames only. The overall compression ratio approaches the ratio of the B-frames themselves (Tab. 1).

The effect of the frame promotion is also clearly observable.

4.3 Decompression time

Decompression times are, due to the algorithm, independent on the actual data sets. Using the data sets of 250 grid points in each dimension and 16bit quantization the current experimental implementation achieves 8, 2, and 3 seconds[1] for I-, P-, and B-frames at AMD Opteron 885 at 2.6 GHz with sufficiently large memory to eliminate the effects of disk I/O. The measured times correspond to the complexity of the decoding formulae of the frame types.

The times may look too large for a single frame, and they are rather far from the target real-time requirement of the haptic simulation, however, they correspond to the size of the data (approx. 100× more than HDTV).[2] However, as the current implementation is a proof of concept only, there is still room for conventional optimization of the numeric code as well as leveraging apparent data parallelism.

[1] The times do not include the expensive final inverse of the μ-law transform, which is necessary in the haptic application on only a few points — current positions of the interacting ligand atoms.

[2] Standard gzip compression takes approximately the same time with signifficantly worse compression ratio of $\sim 90\%$.

4.4 Further quantization of P- and B- frames

We carried experiments with further, more aggressive quantization of residuals of the P- and B-frames. The outcome is very destructive on P-frames because of introducing unacceptable error. The effect is not so apparent on B-frames, however, compression ratio is not improved significantly. Therefore we conclude that further quantization is not usable in the method.

5 Conclusion

The problem we approached, compression of vector field changing in time, exhibits similar properties to audio and video compression, however, it has three spatial dimensions, therefore the involved data size exceed HDTV video by approx. 100 times. We proposed a compression method inspired by both audio and video codecs, and we evaluated its properties.

The achieved compression ratio is about 7–8 % of the original data size. Despite the compression method is lossy, the induced relative error seems to be acceptable, however, the real effect on the haptic simulation still has to be evaluated. The compression ratio and data loss can be traded off by tuning compression parameters, though. Decompression times of the current naïve implementation are rather high, corresponding to the data sizes, however, we foresee room for fairly straightforward improvements by optimization of the numeric code and parallel implementation as well.

Despite the method was designed to compress electrostatic force field of molecular interaction, it does not rely on many of its specific properties. It is generally applicable on similar vector fields in other applications where lossy behaviour is allowed and where the assumptions of high dynamic range of the signal and continuous values hold.

References

1 FLAC. Flac – format, 2008. Online; last access 2009/12/23; available on: `http://flac.sourceforge.net/format.html`.

2 Mat Hans and Ronald W. Schafer. Lossless compresion of digital audio. Technical Report HPL-99-144, November 1999.

3 Esther Kellenberger, Jordi Rodrigo, Pascal Muller, and Didier Rognan. Comparative evaluation of eight docking tools for docking and virtual screening accuracy. *Proteins: Structure, Function, and Bioinformatics*, 57(2):225–242, 2004.

4 Aleš Křenek. *Towards interactive molecular models.* PhD thesis, 2005. Faculty of Informatics, Masaryk University, Brno.

5 Aleš Křenek and Jiří Filipovič. Haptics-assisted docking simulation using virtual coupling, 2010. in preparation.

6 Hiroshi Nagata, Hiroshi Mizushima, and Hiroshi Tanaka. Concept and prototype of protein-ligand docking simulator with force feedback technology. *Bioinformatics*, 18(1):140–146, 2002.

7 Miguel A. Otaduy and Ming C. Lin. Stable and responsive six-degree-of-freedom haptic manipulation using implicit integration. *World Haptics Conference*, pages 247–256, 2005.

8 R. Rice and J. Plaunt. Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Communication Technology*, 19(6):889–897, 1971.

9 Tony Robinson. SHORTEN: Simple lossless and near-lossless waveform compression. Technical Report TR156, December 1994.

Automatic C Compiler Generation from Architecture Description Language ISAC

Adam Husár[1], Miloslav Trmač[1], Jan Hranáč[2], Tomáš Hruška[1], Karel Masařík[1], Dušan Kolář[1], and Zdeněk Přikryl[1]

1 **Brno University of Technology, Faculty of Information Technology**
Bozetechova 2, Brno, Czech Republic
{ihusar, itrmac, hruska, masarik, kolar, iprikryl}@fit.vutbr.cz

2 **ApS Brno, s.r.o.**
Purkynova 93a, Brno, Czech Republic
hranac@aps-brno.cz

Abstract

This paper deals with retargetable compiler generation. After an introduction to application-specific instruction set processor design and a review of code generation in compiler backends, ISAC architecture description language is introduced. Automatic approach to instruction semantics extraction from ISAC models which result is usable for backend generation is presented. This approach was successfully tested on three models of MIPS, ARM and TI MSP430 architectures. Further backend generation process that uses extracted instruction is semantics presented. This process was currently tested on the MIPS architecture and some preliminary results are shown.

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.47

1 Introduction

As semiconductor process node technology advances and allows Moore's law to be still valid, chip designers are faced with a problem how to make a chip that conforms to given performance, power, and cost requirements, but still can be used in many different devices in order to alleviate non-recurring engineering costs (chip design and mask manufacturing) that rise tremendously with each new technology process node.

Electronic System Level (ESL) methodologies try to lower design costs by providing guidelines for SoC (System on Chip) and MPSoC (Multiprocessor SoC) design. One ESL methodology for MPSoC design presented in [5] comprises of several steps, where the most important ones are target application analysis, task partitioning to specific processors, and optimization of such specific processors to suit performance, power and cost requirements. Processors optimized for a certain task are called *Application Specific Instruction-set Processors* (ASIPs).

When an ASIP is designed, the target application is analyzed and hot spots are found. New instructions that accelerate frequent computations are added to the ASIP's instruction set and the application is compiled and analyzed again. This process, often called *compiler-in-the-loop* ASIP design [2], is iteratively repeated until requirements are satisfied. To allow such optimization process, compiler, assembler, and simulator for the current ASIP architecture must be available.

Project Lissom running at the Brno University of Technology approaches this problem by providing an development environment for application-specific instruction processor (ASIP) design and optimization. Using *Architecture Description Language* (ADL) *ISAC* [7], the user can describe both the architecture (instruction set, registers and memories) and

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 47–53

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

microarchitecture (usually pipelined processor implementation). From this description can all the needed tools and hardware description be generated. C language compiler is one of the essential tools and this paper presents a novel approach to higher-level programming language compiler generation from an ADL model.

2 Related Work

2.1 Retargetable Compilers

Retargetable compilers are higher-level programming language compilers that can be adapted to compile for different architectures.

Compilers are usually divided into three components. A frontend first parses input language, performs semantic checks and generates intermediate program representation (IR). Then a midend (also called optimizer or middle-end) is applied and performs mostly target-independent optimizations on IR. Resulting IR is passed to a backend (also called code generator) whose task is to transform the IR into target architecture program.

Examples of retargetable compilers are gcc, LLVM, CoSy, SUIF, lcc, Trimaran, LANCE, and SPAM. Target features may vary substantially and because of flexibility and needs for some target-specific modifications is adapting these listed compilers for a different target requiring extensive compiler expertise and it is up to the tool developers to make the compiler retargetting based on an ADL model as user-retargetable as possible.

2.2 Automatical Compiler Retargeting using an ADL Model

Instruction selection pass is usually performed in a backend as one of the first passes. The purpose is to transform input intermediate representation that uses compiler's IR instructions to a representation with target instructions. Instruction selection pass is generated by so-called code generator generators from some instruction description, where instruction semantics is in form of a tree or DAG (Directed Acyclic Graph) patterns.

Instruction selection pass is the most problematic pass to generate from an ADL model, and in this paper we will focus only on it. For example in LISATek Processor Designer suite, one of the most advanced ASIP design environments used in practice, significant manual effort to create instruction selection patterns from LISA language model is needed [3]. In a recent book C Compilers for ASIPs: Automatic Compiler Generation with LISA [2], they state that to derive instruction selection patterns from instruction behavior in C "is quite difficult, if not impossible". In the first compiler generator version, patterns were described using graphical interface and this caused the semantics information to be stored outside the LISA model in a special format. To partially overcome this problem, a special section SEMANTICS for patterns description was introduced, however, the instruction semantics is still present in the model twice, once in SEMANTICS and once in BEHAVIOR. A similar approach is used in Tensilica TIE [5], where two types of description: one for simulation and for hardware generation, second for compiler generation, are used. Review of other approaches to compiler generation can be found in [8], and also in [2].

We are convinced that a potential ADL user is usually familiar with the C language and to specify instruction behavior using this language is very convenient, better than to learn a new specification language with new syntax and a set of operations. However, as the LISA approach shows, to extract instruction semantics from C language description suitable for compiler generation is difficult and even is such a large project was not solved.

This paper comes with a novel solution that approaches this problem and that allows to extract automatically instruction selection patterns. Further, the process of compiler backend generation that uses such extracted instruction semantics is briefly presented.

3 ISAC Language

The ISAC (Instruction Set Architecture C) language falls into the category of mixed ADLs and allows both to describe architecture and microarchitecture. For purposes of this paper, we will consider only the architectural description. The ISAC language is originally based on the language LISA [1] where we simplified syntax and improved some constructs. Processor architecture consists of register, memory, and instruction set specification.

Description of each instruction consists of textual and binary representation and also of its semantics (behavior). For most existing instruction sets, many of instruction features are similar (like register and immediate operands, or conditional predicates), so for conciseness, the description is hierarchical and it is based on translational context-free grammars. There are two main constructs used to describe the instruction set. The first one is OPERATION, where parts of instruction's syntax, coding, and semantics are described. Construct GROUP is used to describe situations where an instance in an operation can be one of a set of operations or groups. One special group and operation modifier was introduced to the ISAC language because of compiler generation. It is a keyword REPRESENTS and tells that this group or operation is a register operand.

Example description of MIPS architecture instructions ADD and SUB is in fig. 1. Names of sections ASSEMBLER, CODING, and EXPRESSION were abbreviated to ASM, COD, and EXPR, also binary encoding was slightly modified in this example.

```
OPERATION reg REPRESENTS regs {
  ASM { "R" regnum=#U }; COD { regnum=0bx[5] }; EXPR { regnum; } }
OPERATION opc_add { ASM{ "ADD" }; COD{ 0b10 }; EXPR{ 0x2; };}
OPERATION opc_sub { ASM{ "SUB" }; COD{ 0b11 }; EXPR{ 0x3; };}
GROUP opc = opc_add, opc_sub;
OPERATION instr {
  INSTANCE reg ALIAS {rd, rs, rt}; INSTANCE opc;
  ASM { opc rd "," rs "," rt }; // Assembly syntax
  COD { 0b00 rs rt rd opc };    // Binary coding
  BEHAVIOR {                    // Instruction behavior described using C
    switch (opc) {
      case 0x2: regs[rd] = regs[rs] + regs[rt]; break;
      case 0x3: regs[rd] = regs[rs] - regs[rt]; break;
}};}
```

Figure 1 Description of MIPS instructions ADD and SUB in ISAC

To generate a C compiler, we need to extract instruction semantics and syntax for each instruction and then use it to generate compiler backend. We will look at these two steps in the following sections.

```
instr instr__opc_add__reg__reg__reg__, // Name
  %R1 = i32 regop(cl0, 1);             // Semantics
  %R2 = i32 regop(cl0, 2);
  %add = add(%R1, %R2);
  regop(cl0, 0) = i32 %add;,
  "ADD" 0 "," 1 "," 2                  // Syntax
```

Figure 2 Extracted instruction ADD with its semantics and syntax

4 Instruction Semantics Extraction

In the ISAC language is the instruction set described hierarchically using context-free grammars, there is no notion of an instruction present. However, to be able to identify particular instructions in the backend, we need to extract a set of instructions. To get such a set, we simply generate the assembly language from the assembly language grammar obtained from the model. Absence of cycles in this grammar is ensured by the ISAC language compiler, therefore the generated language is finite. Detailed information on grammar extraction from an ISAC model can be found in [6]. For our example in fig. 1, we get a language consisting of two words "ADD reg , reg , reg" and "SUB reg , reg , reg".

We also need unique instruction identification and instruction semantics. To obtain this, we construct a finite automaton with three types of terminals on transitions: operation names, assembly syntax parts, and instance names and parts of semantics in language C. Each path from the starting state to a final state then represents one instruction. For each such path we separately concatenate operation names, assembly terminals, and we also create C code that represents the instruction semantics.

Like this we obtain the instruction syntax and semantics, the only problem is that the form of semantics in the C language we retrieved from the ISAC model is neither suitable for instruction selection pass generation nor for other analyses.

But we can simplify it. We parse this code, then apply optimizations like constant propagation and dead code elimination. Further, memory and register accesses are identified. This way we obtain semantics representation that is usable for instruction selection pass generation. This process is fully automatic and we need no to add to the model information about instructions specific only for compiler generation. This approach overcomes possible inconsistency problems when behavior is described twice in LISA approach (described in subsection 2.2).

The result for our example can be seen in fig. 2. Semantics is described using our SSA-based intermediate representation, auxiliary variables have prefix `%` and `cl0` is an identifier that specifies general-purpose register class.

In this example, instruction semantics is described as a simple DAG with two register input operands on leaves. Register values are added and stored into another register operand. As operations in semantics description can be standard arithmetic, register read/write, and memory load/store operations used. Conditional execution is expressed using operation `if`, and jumps with operation `br`.

5 Retargetable Backend Generation

We have decided to base our work on the open-source LLVM compiler[4]. Only trivial modifications are necessary in the frontend, most of the work involves the backend (which

```
def instr__opc_add__reg__reg__reg__:
   LissomInst<
    (outs cl0:$op0), (ins cl0:$op1, cl0:$op2),     // Operands
    "ADD $op0 , $op1 , $op2",                      // Syntax
    [(set cl0:$op0, (add cl0:$op1, cl0:$op2))]>;   // Selection pattern
```

Figure 3 LLVM instruction description example

generates the actual assembler output).

The largest component of the LLVM backend is *instruction selector*, which converts an input program from a target-independent representation into a lower-level representation that deals with instructions of the target architecture. LLVM uses a tree pattern matching instruction selector, which can take advantage of complex instructions, as long as they generate only one result. The instruction selector is automatically generated from instruction descriptions, they include an expression tree representation of the semantics to match, but it also allows adding C++ code to handle more complex cases.

An example of instruction description that is generated from example in fig. 2 is provided in fig. 3.

LLVM also needs some information about the overall structure of the instruction set. Most important is the *legalization* pass, which modifies the input program to only use operations that are available in the target architecture. Unfortunately LLVM cannot extract the required information from the individual instruction descriptions, so this information is generated separately.

Fuirther LLVM needs to generate some target instructions after instruction selection has finished, notably instructions for moves and memory accesses necessary for register allocation and spilling. These instructions are located by finding instructions matching a specific form of operations, that do not have any unwanted side effects.

Finally, we generate code handling function frames, function calls, parameter passing, and other transformations dependent on the architecture Application Binary Interface (ABI) describing calling conventions and register allocation rules. Means to allow the user to specify the ABI are currently being added to the ISAC language. In absence of such information, the backend generator automatically generates a reasonable ABI by examining the existing instructions, e.g. looking for a “return” or “call” instruction.

6 Results and Future Work

Program that extracts compiler generator information from ISAC model was implemented and tested on architectural models of 2 32-bit general-purpose processors MIPS (MIPS32, Release 1) and ARM (ARMv5) and a 16-bit microcontroller MSP430 from Texas Instruments. MIPS and ARM models describe all basic instructions from their instruction set without any extensions and co-processors, model of MSP430 is complete and describes all the instructions this instruction set provides. Results are shown in table 1. All tests were run on Intel Core2 Quad 9550 @2.83GHz, Fedora 9, x86_64, only one core was used. Semantics extraction program was compiled with gcc 4.4.1 using -O3. Total execution time is an average from 5 runs and the standard deviation was ± 3%.

Each instruction of the ARM architecture can have one of 15 different predicates and may use one of 8 different addressing modes and this is the reason why the extracted instructions count is so high. The behavior of most instructions of the the MSP430 architecture is described

using just one ISAC operation that contains large switches. For each such instruction is lots of code generated and this causes high relative extraction time.

We cannot compare these counts of extracted instructions to other approaches, because in the available publications on related work, intermediate instruction-set forms are neglected, and directly the results of generated compilers are presented.

Table 1 Transformation time and count of generated instructions for ISAC models of MIPS and ARM architectures

	MIPS	ARM	MSP430
ISAC lines	1110	1450	2040
C lines	610	1190	665
Count of extracted instructions	281	5741	718
Extraction time	0.5 s	35.5 s	16.0 s

When creating the ISAC model, the designer must be careful about using correct data types, otherwise unnecessary data type conversions in selection patterns are generated. Inspection of generated patterns can reveal diverse bugs in instruction behavior. A tool that graphically displays extracted patterns was developed and this and this can greatly aid in processor design verification.

Extracted semantics for the MIPS architecture was used to generate MIPS LLVM backend. This backend was first working at the time of writing this paper, therefore we present here only preliminary results for a simple program that calculates the Fibonacci sequence.

This program was compiled for MIPS by compilers GCC 4.4.1, and CLANG 1.0 with LLVM 2.8 (CLANG is a frontend that generates intermediate representation for LLVM). Input for our generated backed was obtained by compiling source code with CLANG 1.0 and then optimized (for -O3) with LLVM 2.8 optimizer. Resulting assembly code was then assembled and simulated using tools automatically generated from the MIPS ISAC model (e.g. [7]). Cycle counts needed to execute the program are shown in table 2.

Table 2 Preliminary results for backend generator for the MIPS architecture, values are cycle counts needed to simulate compiled program that calculates the Fibonacci sequence

	GCC	LLVM	Backend generated from the ISAC model
No optimizations (-O0)	1991	2200	1416
All optimizations (-O3)	508	506	913

Current plans for the future are: to allow the user to specify ABI, support for predicated execution, arbitrary bit-width integral data types, floating and vector (SIMD) variables and operations. Also we will improve the backend generator according to semantics description extracted from ARM, MSP430, and other models.

7 Conclusion

This paper presents an approach to higher-level language compiler generation. Backend is the part of compiler, where most of target-specific transformations is done and to accelerate application-specific processor architecture development, we need to generate compiler backend as automatically as possible.

First, architecture description language ISAC is briefly presented. Further, translation from ISAC architecture model to the compiler generation model is described. This approach overcomes problems caused by possible architecture model inconsistencies when one type of description is used for simulation and hardware generation and another type is used for compiler generation as is in similar projects usual. Inspection of extracted patterns can also point to some bugs that may be present in the architecture model and leads the user to use exact data types. Usage of exact data types also results in a model usable for efficient hardware generation.

Instruction semantics extraction from ISAC is fully automatic and it was tested on architectures MIPS, ARM, and MSP430. Extracted information was used to generate a backend for the MIPS architecture and some preliminary results were presented.

Acknowledgments

This research was supported by the grants of MPO Czech Republic FR-TI1/038, by the Research Plan MSM No. 0021630528, by the doctoral grant GA CR 102/09/H042, by the BUT FIT grant FIT-SS-10-1, and by the European project SMECY.

References

1 Andreas Hoffmann, Heinrich Meyr, and Rainer Leupers. *Architecture Exploration for Embedded Processors with Lisa*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

2 Manuel Hohenauer and Rainer Leupers. *C Compilers for ASIPs: Automatic Compiler Generation with LISA*. Springer Publishing Company, Incorporated, 2009.

3 Paolo Ienne and Rainer Leupers, editors. *Customizable Embedded Processors*. Morgan Kaufmann, 2007.

4 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

5 Steve Leibson. *Designing SOCs with Configured Cores: Unleashing the Tensilica Xtensa and Diamond Cores (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

6 Roman Lukáš, Tomáš Hruška, Dušan Kolář, and Karel Masařík. Two-Way Deterministic Translation and Its Usage in Practice. In *Proceedings of 8th Spring International Conference - ISIM'05*, pages 101–107, 2005.

7 Karel Masařík, Tomáš Hruška, and Dušan Kolář. Language and Development Environment For Microprocessor Design Of Embedded Systems. In *Proceedings of IFAC Workshop on Programmable Devices and Embedded Systems PDeS 2006*, pages 120–125. Faculty of Electrical Engineering and Communication BUT, 2006.

8 Prabhat Mishra and Nikil Dutt, editors. *Processor Description Languages*. Morgan Kaufmann, 2008.

Efficient Computation of Morphological Greyscale Reconstruction

Pavel Karas

Centre for Biomedical Image Analysis
Faculty of Informatics
Masaryk University Brno

Abstract

Morphological reconstruction is an important image operator from mathematical morphology. It is very often used for filtering, segmentation, and feature extraction. However, its computation can be very time-consuming for some input data. In this paper we review several efficient algorithms to compute the reconstruction, and compare their performance on real 3D images of large sizes. Furthermore, we propose a GPU implementation which performs up to 15 × faster than the CPU methods. To our best knowledge, this is the first GPU implementation of the morphological reconstruction, described in literature.

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.54

1 Introduction

Mathematical morphology is a theory for analysis and processing spatial structures in images. Morphological methods can be used for image pre-processing and for image analysis [9, 14, 16].

Morphological reconstruction is an advanced approach to image analysis. It can be used for various applications, such as filtering, segmentation, and feature extraction [18], image and video compression [13], remote sensing [15], and biomedical image analysis [12]. In image segmentation, the reconstruction is often used for pre-processing, to avoid over-segmentation [5, 8].

To compute the morphological reconstruction, several sequential and FIFO-based algorithms were proposed [11, 18]. To our best knowledge, no GPU implementation of the morphological reconstruction has been described in literature. Eidheim et al. [6] proposed a GPU implementation of basic morphological operations, such as dilation and erosion. These algorithms are easy to implement in parallel as described in [3]. Jivet et al. [10] implemented the morphological reconstruction on a dedicated FPGA hardware using the iterative computation of the geodesic dilation. In this paper, we adopt the sequential algorithm [18] with the reduced number of iterations and propose a parallel GPU-based implementation. We compare its performance with several algorithms executed on CPU.

1.1 Notations

In the following text, an n-dimensional image f is considered a mapping from a finite subset $D_f \subset \mathbb{Z}^n$ into a set I of image values. I is usually a finite discrete set of m levels $\{0, 1, \ldots, m-1\}$. The discrete grid $G \subset \mathbb{Z}^n \times \mathbb{Z}^n$ provides the neighbourhood relationship between pixels: p is a neighbour of q if and only if $(p, q) \in G$. Depending on a particular application, various grids can be used; in 2-D case, 4-, 6-, or 8-connectivity are the most common examples.

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 54–61

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.2 Definition of Morphological Reconstruction by Dilation

Before we define the reconstruction by dilation, we define the geodesic dilation first. We use the same definition as in [15]. Let f, g be two images fulfiling the following properties:

- $\mathcal{D}_f = \mathcal{D}_g$, i.e., both images share the same definition domain,
- $f(p) \leq g(p), \forall p \in \mathcal{D}_f$, i.e., f is smaller or equal to g in all pixels

We call f the *marker* image and g the *mask* image and define the *geodesic dilation* of size 1 as follows:

$$\delta_g^{(1)}(f) = \delta^{(1)}(f) \wedge g, \tag{1}$$

where $\delta^{(1)}$ denotes the elementary dilation (i.e., dilation with the smallest non-trivial structure element of the used connectivity) and the $\wedge$ operator denotes point-wise minimum. The geodesic dilation of size $n > 1$ is obtained by performing n successive geodesic dilations:

$$\delta_g^{(n)}(f) = \delta_g^{(1)}\left[\delta_g^{(n-1)}(f)\right]. \tag{2}$$

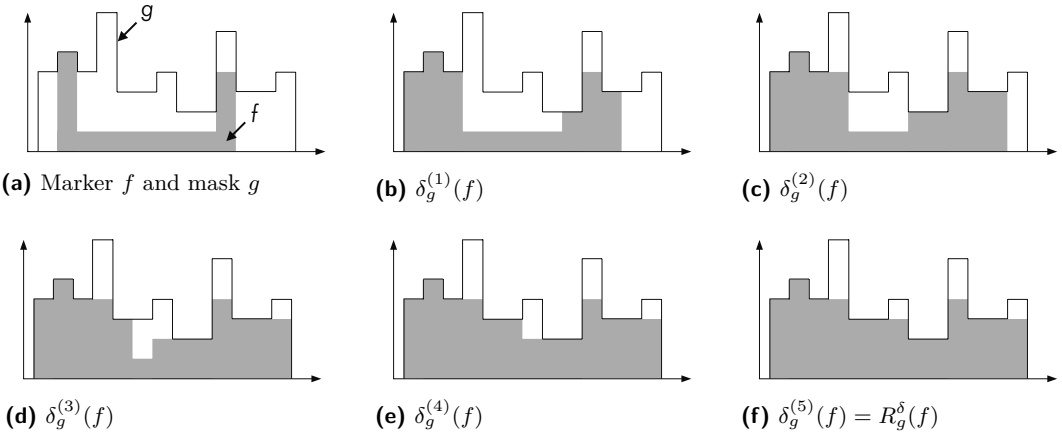
The *reconstruction by dilation* is defined as the geodesic dilation iterated until stability:

$$R_g^{\delta}(f) = \delta_g^{(i)}(f), \tag{3}$$

where i is such that $\delta_g^{(i+1)}(f) = \delta_g^{(i)}(f)$. The reconstruction by dilation of a 1-D signal is illustrated in Fig. 1.

In our application of biomedical image analysis, we process 3-D grayscale images. Therefore, we describe our implementations for the 3-D images with the 6-connectivity.

The paper is organized as follows: First, we review three algorithms for computing the morphological reconstruction, described in literature [18]. Second, we describe our GPU implementation. Finally, we analyze and compare the performance of all implementations on several 3D images from our field.



(a) Marker f and mask g **(b)** $\delta_g^{(1)}(f)$ **(c)** $\delta_g^{(2)}(f)$

(d) $\delta_g^{(3)}(f)$ **(e)** $\delta_g^{(4)}(f)$ **(f)** $\delta_g^{(5)}(f) = R_g^{\delta}(f)$

Figure 1 Reconstruction by dilation R^{δ} of a 1-D signal g from a marker signal f.

2 Methods

2.1 Existing Algorithms

Standard technique

The reconstruction by dilation can be computed directly from its definition (3). Even though the iterations can be performed efficiently using van Herk/Gil-Werman algorithm [7, 17], an enormous number of iterations is required to converge. Therefore, we do not consider this algorithm in our paper in the performance evaluation.

Sequential reconstruction (SR)

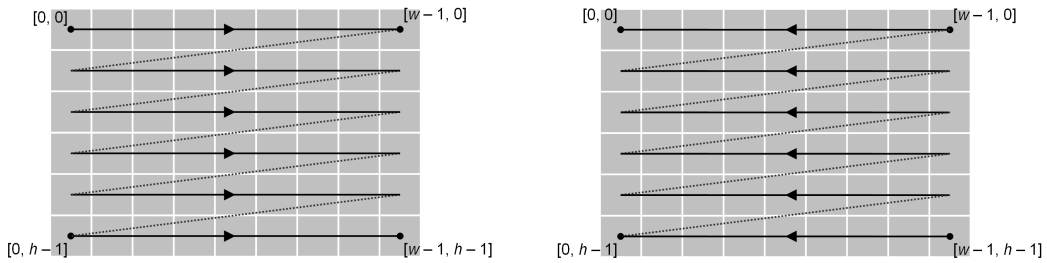
This algorithm was proposed to reduce the number of iterations [18]. The image is scanned in a predefined order and the information is propagated throughout the image. First, the image is scanned in the raster order—see Fig. 2a. Subsequently, it is scanned in the anti-raster order—Fig. 2b. The scans are repeated until convergence. The computation is performed "in-place" in the marker image.

Hybrid reconstruction algorithm (HRA)

HRA does not yield multiple iterations, thus, the computation time is further reduced [18]. It has two phases: a sequential and a FIFO phase. The former uses the previous sequential algorithm to execute a single iteration. During the anti-raster scan, pixels of regional maxima are put into a queue. In the latter phase, the pixels are read from the queue and their neighbourhood pixels are examined. If the information is propagated to the neighbourhood pixels, they are put into the queue. Once the queue is empty, the computation is complete.

2.2 GPU Implementation (*SR_GPU*)

Our GPU implementation is a modified version of the *SR* algorithm described in Section 2.1 and Fig. 3. It was written in the CUDA parallel programming model [2]. For the flowchart of the *SR_GPU* algorithm refer to Fig. 3. The raster and anti-raster scans are performed in each dimension, separately. Thus, each iteration requires 4 or 6 passes for a 2-D or 3-D image, respectively. The raster scans are executed by CUDA kernels called *x_Forward*, *y_Forward*, and *z_Forward*; the anti-raster scans are performed by kernels called *x_Backward*, *y_Backward*, and *z_Backward*. Since the Forward and Backward kernels are analogous, only the Forward variants will be described.



(a) Raster scan **(b)** Anti-raster scan

Figure 2 Scanning patterns in a 2-D image of size $w \times h$ pixels.

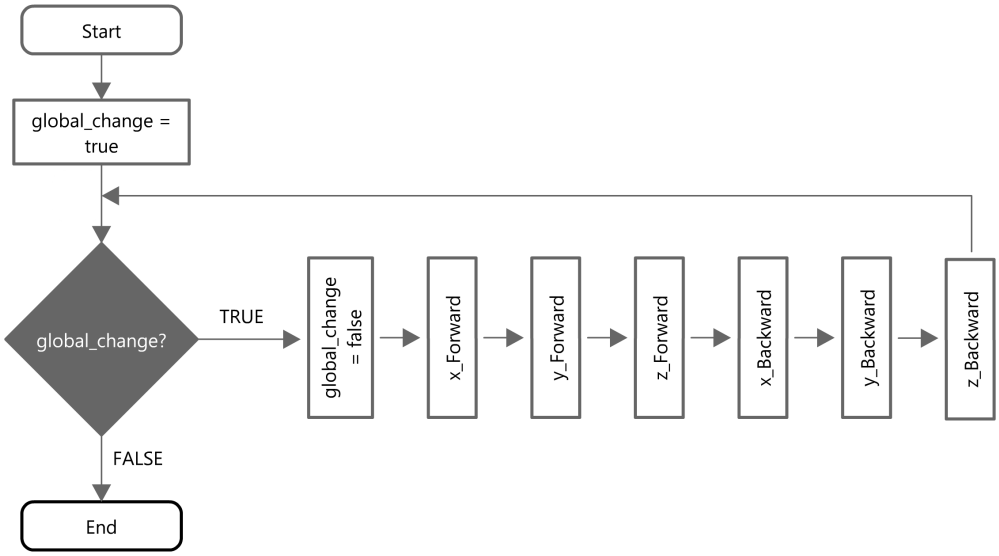


Figure 3 Flowchart of the *SR_GPU* algorithm for the morphological reconstruction. The functions, called *x_Forward*, *y_Forward*, *z_Forward*, *x_Backward*, *y_Backward*, and *z_Backward*, provide image scans. The *global_change* variable indicates changes in the marker image.

y_Forward

The y_Forward kernel is executed by $N_x N_z$ threads, where N_x, N_z are the sizes of the input images in x and z dimension, respectively. Except the input images *marker* and *mask* and the variable called *global_change* stored in the global memory, all the variables can be stored in registers. Since the threads are regularly distributed across the x and z dimensions of the images, the accesses to the global memory are naturally coalesced [2], achieving the maximum bandwidth.

z_Forward

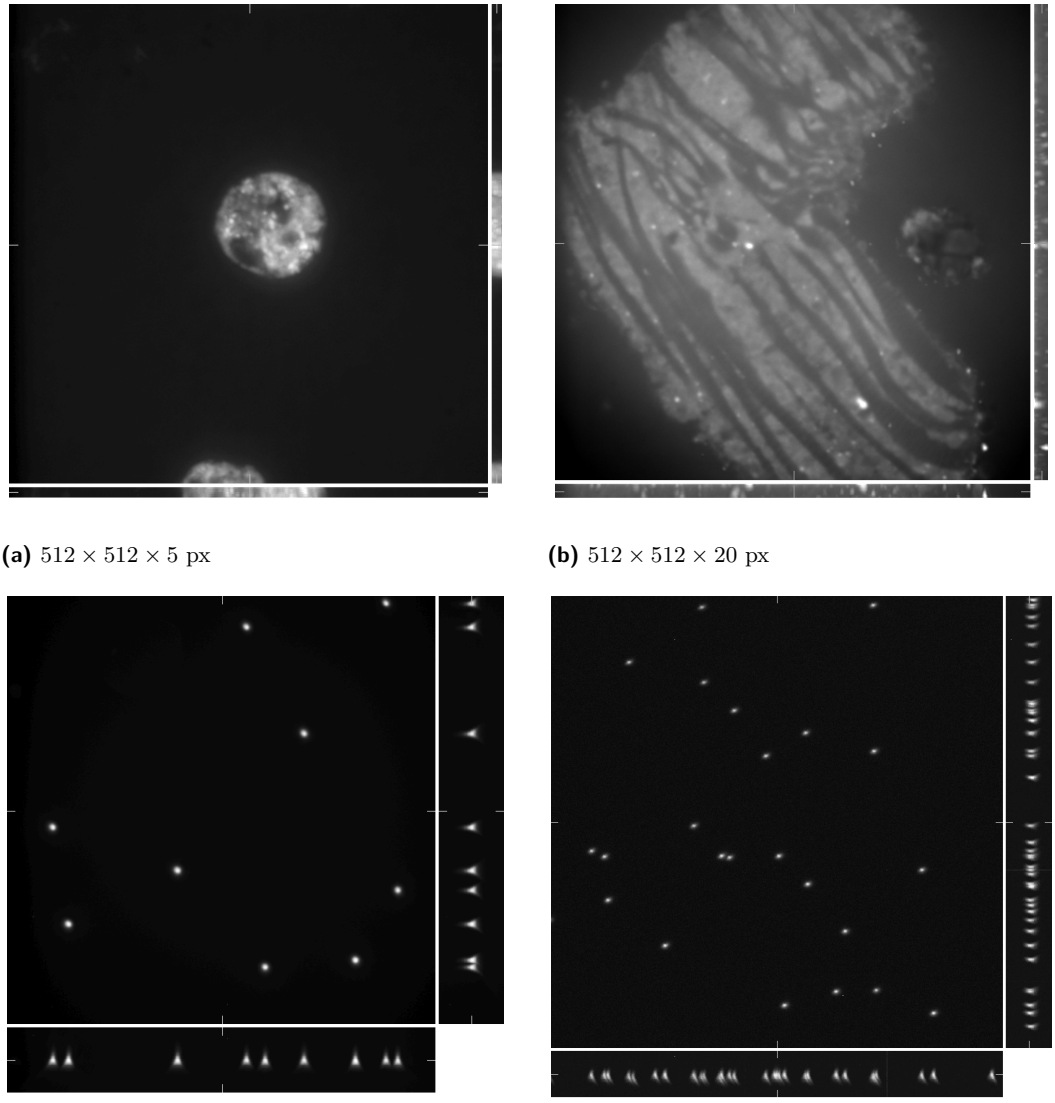
The z_Forward kernel is analogous to the y_Forward kernel.

x_Forward

The x_Forward kernel is executed by $N_y N_z$ threads. Since the threads are now distributed across the y and z dimensions of the images, we cannot use the same approach as above to achieve optimum access to the global memory. Therefore, we pre-load data and compute results in the shared memory [2]. Afterwards, the data is written back to the global memory. Only few threads of a block perform the computation itself, so this approach may seem to be less efficient. However, the GPU thread scheduler can switch between warps, thus overlapping the arithmetic operations and memory accesses and hiding the global memory latency.

Stopping criterion

The scans are repeated until convergence, much like in the classic *SR* algorithm. The test of convergence is performed by inspecting the *global_change* variable after each scan. Gath-



(a) 512 × 512 × 5 px

(b) 512 × 512 × 20 px

(c) 512 × 512 × 100 px

(d) 1300 × 1030 × 80 px

Figure 4 Input images.

ering information from all threads to one output variable generally requires the *reduction* kernel [1]. However, in this case, only a boolean-type information is needed, thus, the reduction can be avoided. Before each scan, *global_change* is set to *false*. Iff a thread performs the first change in the marker image, it assigns *global_change* to *true*. This approach also avoids write-before-read conflicts.

3 Results

The performance of the three algorithms, namely *SR*, *HRA*, and *SR_GPU*, was compared on four real 3-D images from confocal microscopy (Fig. 4). These images were taken as the input mask image g.

Two different approaches to create the marker image f were chosen. First, all but one of the pixels of the marker image are set to zero. The non-zero pixel was selected to be the one with the maximum value in the mask image and its value was set to the same value. Second, the values of pixels in the marker image were set to the values of those in the mask image, decreased by a constant h:

$$f(p) = \max\{g(p) - h, 0\}. \tag{4}$$

The morphological reconstruction with the marker image defined as above is often called the *HMAX* transform in literature [16].

The implementations were tested on a workstation with an Intel Core2 Quad Q6600 2.4 GHz CPU, 8 GB DDR2 RAM, and a GeForce GTX 470 GPU with 448 SPs and 1280 MB of GDDR5 memory.

3.1 Morphological Reconstruction With a Simple Marker Image

The results of the first experiment are summarized in Table 1. The computation times are in seconds, the data-transfer overhead for the GPU implementation is included.

It is obvious that results of both the *SR* and *SR_GPU* algorithm strongly depend on the number of iterations needed to complete the computation. The number of iterations depends strongly on both the image dimensions and the image content. The queue-based *HRA* algorithm does not yield such iterations and converges significantly faster. However, the GPU implementation is faster almost in all cases, due to higher performance and memory bandwidth of the graphics hardware. The image (d) is the only case where the *HRA* algorithm on CPU performs better, since the number of iterations is extremely high (661). In other cases, the GPU implementation achieves up to 15 × speedup over the *HRA* algorithm.

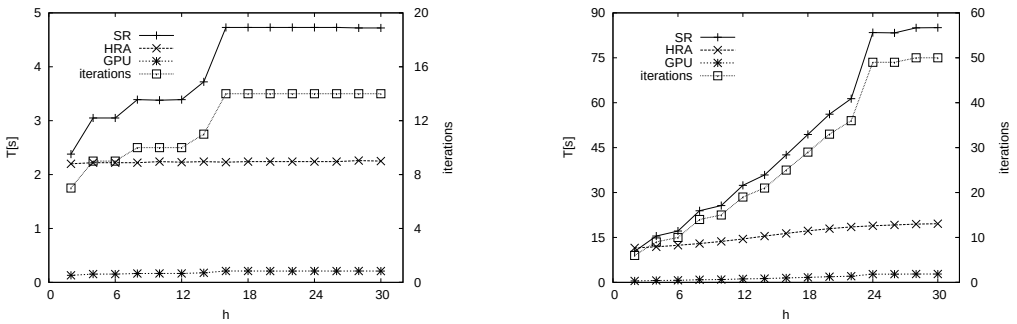
3.2 HMAX Transform

The results of the second experiment are summarized in Table 2. They are very similar to those in the previous experiment, however, the number of iterations is generally lower.

We also analysed the dependancy of both the computation time and the number of iterations on the h parameter. The results for two selected images are shown in Fig 5. As expected, the dependancy is strong for the *SR* and the *SR_GPU* algorithms, while there is almost no dependancy for the *HRA* algorithm. However, the computation time for the *SR_GPU* algorithm does not grow so fast with increasing the number of iterations, because with the higher computation time, the effect of the data-transfer overhead is reduced.

Table 1 Morphological reconstruction with a simple marker image. In the columns 2, 3, and 4, computation times in seconds are presented. The column 5 shows the number of iterations needed to complete the computation in *SR* and *SR_GPU*. In the last column, the speedup achieved by the *SR_GPU* algorithm is presented.

Image	*SR* [s]	*HRA* [s]	*SR_GPU* [s]	iterations	speedup
(a)	8.02	2.26	0.22	30	10.3
(b)	18.74	6.17	0.51	17	12.1
(c)	280.23	64.57	4.16	51	15.5
(d)	> 2 hours	98.97	135.13	661	0.7



(a) Image (b) **(b)** Image (c)

Figure 5 Computation time and number of iterations for the HMAX transform on two selected images.

4 Conclusion

In this paper we proposed a GPU implementation for the morphological reconstruction and compared its performance with two CPU algorithms. The results showed that graphics hardware offers good speedup and is able to perform significantly faster than the optimized CPU algorithm in most cases.

By optimizing our GPU implementation, further speedup could be achieved. The main issue is the high number of iterations for some input data. By implementing the optimized *HRA* algorithm, this could be avoided, but FIFO-based algorithms are not generally good candidates for GPU acceleration. In our future work, we will study possibilities of adopting the *HRA* algorithm for GPU.

The CPU implementations can be improved, too, for example, by utilizing multiple CPU cores. However, in the case of the faster queue approach this would require a challenging effort. The performance of the *SR* algorithm strongly depends on the number of cache misses and could be also improved by using cache-efficient matrix transpositions [4]. This is the subject of our future work.

Table 2 HMAX transform with the parameter $h = 10$. In the columns 2, 3, and 4, computation times in seconds are presented. The column 5 shows the number of iterations needed to complete the computation in *SR* and *SR_GPU*. In the last column, the speedup achieved by the *SR_GPU* algorithm is presented.

Image	*SR* [s]	*HRA* [s]	*SR_GPU* [s]	iterations	speedup
(a)	0.57	0.56	0.06	7	9.3
(b)	3.39	2.23	0.17	10	13.1
(c)	25.63	13.73	0.94	15	14.6
(d)	1414.25	71.73	40.37	204	1.8

Acknowledgments

This work has been supported by the Ministry of Education of the Czech Republic (Projects No. MSM-0021622419, No. LC535 and No. 2B06052).

References

1 CUDA™ SDK Code Samples 3.1. `http://developer.nvidia.com/object/cuda_sdk_samples.html`, Jun 2010.

2 NVIDIA GPU Computing Developer Home Page. `http://developer.nvidia.com/object/gpucomputing.html`, Jun 2010.

3 Thomas Bräunl, Stefan Feyrer, Wolfgang Rapf, and Michael Reinhardt. *Parallel Image Processing*. Springer, 2001.

4 S. Chatterjee and S. Sen. Cache-efficient matrix transposition. pages 195–205, 2000.

5 Xiaowei Chen, Xiaobo Zhou, and S.T.C. Wong. Automated segmentation, classification, and tracking of cancer cell nuclei in time-lapse microscopy. *IEEE Transactions on Biomedical Engineering*, 53(4):762–766, 2006.

6 O.C. Eidheim, J. Skjermo, and L. Aurdal. Real-time analysis of ultrasound images using GPU. *International Congress Series*, 1281:284–289, 2005. CARS 2005: Computer Assisted Radiology and Surgery.

7 J. Gil and M. Werman. Computing 2-D min, median, and max filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(5):504–507, may. 1993.

8 K. Haris, S. N. Efstratiadis, N. Maglaveras, and A. K. Katsaggelos. Hybrid image segmentation using watersheds and fast region merging. *IEEE Transactions on Image Processing*, 7(12):1684–1699, 1998.

9 Bernd Jähne. *Digital Image Processing*. Springer, 6th edition, 2005.

10 Ioan Jivet, Alin Brindusescu, and Ivan Bogdanov. Image contrast enhancement using morphological decomposition by reconstruction. *WSEAS Trans. Cir. and Sys.*, 7(8):822–831, 2008.

11 Kevin Robinson and Paul F. Whelan. Efficient morphological reconstruction: a downhill filter. *Pattern Recognition Letters*, 25(15):1759–1767, 2004.

12 Pekka Ruusuvuori, Tarmo Aijo, Sharif Chowdhury, Cecilia Garmendia-Torres, Jyrki Selinummi, Mirko Birbaumer, Aimee Dudley, Lucas Pelkmans, and Olli Yli-Harja. Evaluation of methods for detection of fluorescence labeled subcellular objects in microscope images. *BMC Bioinformatics*, 11(1):248, 2010.

13 P. Salembier, P. Brigger, J.R. Casas, and M. Pardas. Morphological operators for image and video compression. *IEEE Transactions on Image Processing*, 5(6):881–898, 1996.

14 Jean Serra. *Image Analysis and Mathematical Morphology*. Academic Press, Inc., Orlando, FL, USA, 1983.

15 P. Soille and M. Pesaresi. Advances in mathematical morphology applied to geoscience and remote sensing. *IEEE Transactions on Geoscience and Remote Sensing*, 40(9):2042–2055, 2002.

16 Pierre Soille. *Morphological Image Analysis: Principles and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

17 Marcel van Herk. A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels. *Pattern Recognition Letters*, 13(7):517–521, 1992.

18 Luc Vincent. Morphological grayscale reconstruction in image analysis: Applications and efficient algorithms. *IEEE Transactions on Image Processing*, 2:176–201, 1993.

On Reliability and Refutability in Nonconstructive Identification

Iļja Kucevalovs

Institute of Mathematics and Computer Science,
University of Latvia, Raina bulv. 29, Riga, Latvia
ilja.kucevalovs@lais.lv

Abstract

Identification in the limit, originally due to Gold [10], is a widely used computation model for inductive inference and human language acquisition. We consider a nonconstructive extension to Gold's model. Our current topic is the problem of applying the notions of reliability and refutability to nonconstructive identification. Four general identification situations are defined and two of them are studied. Thus some questions left open in [13] are now closed.

Keywords and phrases inductive inference, identification, reliability, refutability, nonconstructive computation

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.62

1 Introduction

The computational model of inductive inference known as identification (also: identification in the limit, algorithmic learning, etc.), introduced by Gold [10], and its many variations have been widely studied. The reader is encouraged to refer to [20] and [14] for detailed surveys on applying Gold's model to learning recursive functions and recursive languages from positive data, respectively.

Gold's original model deals with some abstract computational device which makes guesses about some object it has as an input. That device, also referred to as inductive inference machine (IIM), is usually said to identify a class of objects if it identifies (i.e. correctly guesses) any object within that class. However, in the general case the IIM behaviour on objects not from the class in question is not specified.

This issue is dealt with in the "reliable identification" model, which is due to Blum and Blum [2], Minicozzi [16], and later Sakurai [19]; "refutable identification", first considered by Mukouchi and Arikawa [17], is a strengthening of the reliable model. Both these models prohibit the IIM to output any sequence which could be a correct sequence of guesses for some class member (if the given object is not such a member), and the refutable model specifies the IIM to explicitly refute any non-member.

In the thesis [13], Gold's model was extended with nonconstructive computational methods, which allow the IIM to utilize some additional information. There is, however, a certain difference between the nonconstructive identification and the traditional "identification with additional information" (see e.g. [4, 7, 11, 12]). We briefly introduce the two main distinctions. First, all the three nonconstructive identification models (K, S and F) are given on a general level, instead of defining a separate criterion for each particular situation (as it is usually being done). Second, these models are specially constructed so that a trivial help (i.e. supplying the desired answer) would not be possible.

Allowing additional information for identification introduces the following dilemma: Should we always assume that the help the IIM gets is correct (and if not, should we refute

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 62–68

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

this help)? Note the obvious similarity between this problem and the one stated above, solved by introducing reliability for identification. That motivated the author to consider two levels of reliability and refutability and, correspondingly, four different identification situations in nonconstructive identification.

In the current paper, we show a class of non-recursive functions that is not identifiable without any additional information, but is nonconstructively identifiable with the following properties. On the one hand, the help information it is identifiable with can be biased by a function that grows to infinity; on the other hand, it is refutably identifiable (utilizing, however, another kind of help).

To our knowledge, the problems of such type were neither previously solved nor considered at all.

2 Preliminaries

In this section, we briefly introduce the notions used in this paper. Notions from recursive function theory not explained here are treated in e.g. Rogers' textbook[18]; a more brief introduction into recursion theoretic notions can be found in Gallier and Hicks' online books [8, 9]. The notion of Kolmogorov complexity is explained in detail in [15].

N denotes the set of nonnegative integers $\{0, 1, 2, ...\}$. N^+ denotes the set of positive integers $\{1, 2, 3, ...\}$. R denotes the set of all total recursive functions.

$\langle x_0, x_1, ..., x_k\rangle$ is an ordered tuple of elements x_0, x_1, ..., x_k (in that order). If X is such a tuple, we refer to each its kth element as X_k. Given some universal set U, we write $\langle U^k\rangle$ to denote a set of all tuples of some length k over U. $\langle U^*\rangle \equiv \cup_{k\in U}\langle U^k\rangle$, i.e. the set of all finite length tuples over U. $\langle U^\infty\rangle$ denotes the set of all infinite length tuples over U. $\langle U\rangle \equiv \langle U^*\rangle \cup \langle U^\infty\rangle$, i.e. the set of all possible tuples over U, of both finite and infinite length.

$\forall^\infty, \exists^\infty, \exists!$ denote "for all but finitely many", "there exist infinitely many", "there exists a unique", respectively.

For an object x, $l(x)$ denotes the length of the binary presentation of x. For a set X, $d(X)$ is its cardinality and $\bar{X}$ is its complement. For a sequence X, $l(X)$ is its length. For a sequence X, $X[n]$ denotes its initial segment of length n. For a total function f, $f[n]$ denotes the initial segment of its graph. We write $f(x_0)\downarrow$ and $f(x_0)\uparrow$ to denote that f is defined (undefined) on x_0.

Given a class U, we call any partial recursive function $\varphi : N \to U$ a numbering for U. We say that U is *defined* in φ (written: $\varphi(U)\downarrow$), iff $\forall u \in U : (\exists n \in N)[\varphi_n = u]$. Talking about identification of some U in some φ, we always assume $\varphi(U)\downarrow$.

Having fixed some universal Turing machine M_{uni}, by (plain) Kolmogorov complexity of some object u we call the value

$$C(u) = \min\{l(p) : M_{uni}(p) = u\} \tag{1}$$

In this paper, only the so-called "plain" Kolmogorov complexity will be used (for more details see [15]), and our results will rely on one fixed M_{uni}.

A tuple $J \in \langle N^\infty\rangle$ is a BC-sequence for some object u in some numbering $\varphi = \varphi_0, \varphi_1, \varphi_2, \ldots$ (written: $J \in BC(u, \varphi)$), iff $\forall^\infty n \in N : \varphi_{J_n} = u$.

A tuple $J \in \langle N^\infty\rangle$ is an EX-sequence for some u in some φ (written: $J \in EX(u, \varphi)$), iff $J \in BC(u, \varphi) \wedge (\forall^\infty n \in N : J_n = J_{n+1})$.

A tuple $J \in \langle N^\infty\rangle$ is a FIN-sequence for some u in some φ (written: $J \in FIN(u, \varphi)$), iff $J \in EX(u, \varphi) \wedge (\forall n \in N : J_n = J_{n+1})$.

We say that FIN, EX, BC are identification criteria. We say that $I : U \to \{\langle N \rangle\}$ is an (information) presentation of a class U. We say that I is *injective*, iff $\forall u, v \in U : (u \neq v) \Rightarrow (I(u) \cap I(v) = \varnothing)$. We assume that any presentation I we deal with is injective. We say that I is *unambiguous*, iff $\forall u \in U : d(I(u)) = 1$.

A presentation of a recursive function f is its graph, i.e. pairs $\langle x, f(x) \rangle$, which can be reduced to simply a sequence of values $f(0), f(1), f(2), \ldots$ in case we deal with total functions only (like in this paper). We assume, without a loss of generality [10], that a graph of any function f is always input in its natural order, i.e. $f(0), f(1), f(2)$, and so on. Clearly, a function graph is an injective presentation.

An *inductive inference machine* (IIM) is an abstract device that receives positive integers from time to time and generates positive integers from time to time. If M is some IIM, by $M(X) = Y$ we denote "Y is the output M writes having received some input X".

Given some identification criterion $X : X \in \{FIN, EX, BC\}$, some IIM M, some object u and its presentation I and some numbering φ, we say M X-identifies u from I in φ iff $M(I(u)) \in X(u, \varphi)$. Obviously, if M EX-identifies some u, it BC-identifies it; if M FIN-identifies u, it EX-identifies it.

Given some identification criterion X, some IIM M, some class of objects U, some presentation I and some numbering φ, we say that M X-identifies U from I in φ iff M X-identifies every $u \in U$ from I in φ.

Given some identification criterion X, some IIM M, some class of objects U, some presentation I and some numbering φ, we say that M reliably X-identifies U from I in φ iff the following holds:

1. $\forall u \in U : M(I(u)) \in X(u, \varphi)$;
2. $\forall u \notin U : M(I(u)) \notin X(u, \varphi)$.

Given some additional refutation symbol #, we say that M refutably X-identifies U from I in φ iff the following holds:

1. $\forall u \in U : M(I(u)) \in X(u, \varphi)$;
2. $\forall u \notin U : M(I(u))_{l(M(I(u)))} =$"#".

We say that U is (reliably, refutably) X-identifiable from I in φ iff there exists an IIM that (reliably, refutably) X-identifies U from I in φ.

3 Nonconstructive identification

There are several definitions of nonconstructive identification [13]. Here we consider one of them, the so-called F-nonconstructivity, which is very similar to identification given the upper bound on the program size (see e.g. [4, 12]). However, the definition given here does not limit identification to function or language learning, or to any other particular type of learning.

▶ **Definition 1.** Given some identification criterion X, some IIM M, some class U, some presentation I and some numbering $\varphi : \varphi(U) \downarrow$, we say that M F-nonconstructively X-identifies U from I in φ with amount of nonconstructivity $p(n)$, iff $\forall n \in N : \exists m \in N$ s.t. the following holds:

1. $m \leq p(n)$;
2. $\forall u \in U \cap \{\varphi_i | i \leq n\} : M(\langle I(u), m \rangle) \in X(u, \varphi)$.

Further in the text "nonconstructive", "nonconstructively" etc. is also referred to as "NK".[1] Any identification model without additional information is called "constructive".

4 Application of reliability and refutability

One can consider two levels of applying reliability and/or refutability to nonconstructive identification:

1. Reliable / refutable identification;
2. Reliable / refutable nonconstructivity.

On the first level, one deals with the usual problem of reliability and refutability: Which classes in which numberings can be identified so that input of a non-member presentation would not result in misleading "identification" of it. Nonconstructive methods are used to assist reliability or refutability.

On the second level, the problem is the following: Which is that nonconstructive identification model (if it exists), such that not only correct, but also incorrect nonconstructive information helps identifying some class in some numbering in compliance with some criterion. By saying "helps" we mean that no constructive identification would be possible in that case.

Thus, in accordance with [13], the following four situations are defined:

1. NK-X (the usual nonconstructive identification model);
2. NK-R-X (reliable / refutable models utilizing nonconstructivity);
3. R-NK-X (nonconstructive models which are required to work correctly with incorrect help);
4. R-NK-R-X (reliable / refutable nonconstructive models which are required to work correctly with incorrect help).

We will also use the situation names to denote the corresponding inferring power classes. That is, for a situation Z, the class Z is the set of classes which are Z-identifiable.

In [13], only the NK-X situation was studied. It was noted (and some examples were given) that even emptiness (non-emptiness) of the classes NK-R-X and R-NK-X can be tedious tasks to solve (while proving the R-NK-R-X case would obviously close both these questions). Below we prove non-emptiness of NK-R-X and R-NK-X, leaving out R-NK-R-X.

The above definition of F-nonconstructivity is modified in order to properly define the R-situations.

▶ **Definition 2.** Given some total $E : N \to N$ s.t.

$$\lim_{n\to\infty} E(n) = \infty \tag{2}$$

as well as some identification criterion X, some IIM M, some class U, some presentation I and some numbering $\varphi : \varphi(U)\downarrow$, we say that M E-reliably F-nonconstructively X-identifies U, iff $\forall n \in N : \exists m \in \langle N\rangle$ s.t. the following holds:

1. $\forall u \in U \cap \{\varphi_i | i \leq n\}, \forall j \in N : M(\langle I(u), m_j\rangle) \in X(u, \varphi)$;
2. $\forall u \in U \cap \{\varphi_i | i \leq n\} : \lim_{j\to\infty} M(\langle I(u), m_j \pm E(j)\rangle) \in X(u, \varphi)$.

[1] From Latvian "nekonstruktīvs" (nonconstructive). This is being done in accordance with the original thesis [13], which was written in Latvian.

5 Results

▶ **Theorem 3.** *There exists a class U and a numbering W such that the following properties hold:*

1. *U is not constructively identifiable in W;*
2. *U is F-nonconstructively FIN-identifiable in W (NK-FIN);*
3. *U is F-nonconstructively refutably FIN-identifiable in W (NK-Ref-FIN);*
4. *U is reliably F-nonconstructively FIN-identifiable in W (Rel-NK-FIN).*

Proof. The NK-FIN part is immediate from Rel-NK-FIN.

Consider the total functions $h, m : N \to N$ defined as follows:

$$h(x) = \begin{cases} C(1024), & x = 0 \\ \min\{n \in N \mid (n > h(x-1)) \wedge (C(n) > C(h(x-1)))\}, & x > 0 \end{cases} \tag{3}$$

$$m(x) = \min\{C(n) \mid n \in N \wedge n \geq x\} \tag{4}$$

It is obvious that both such functions do exist; however, neither h [13] nor m [15] can be recursive. Moreover, m, despite being unbounded from above, grows slower than any computable function [15].

Let $\langle p_0, p_1, ...\rangle$ be a growing sequence of all the prime numbers starting with $p_0 = 3$. For every $k \in N$, we define

$$f_k(x) \equiv h((p_k)^x) \tag{5}$$

The numbering W is defined as follows:

$$w_i = \begin{cases} f_k, & \exists k \in N, j \in N \setminus [0; k-1], \\ & \quad n \in N \cap \left[h(j) - \lfloor \frac{m(j)}{2} \rfloor; h(j) + \lfloor \frac{m(j)}{2} \rfloor\right] : \\ & \quad f_k(n) = i \\ h, & otherwise \end{cases} \tag{6}$$

We now briefly explain the idea of the above construction. Each function f_k outputs h values from arguments taken from powers of the k-th prime. That is, $\text{range}(f_i) \cap \text{range}(f_j) = \{1\}$ for every natural $i \neq j$. Moreover, range of every f_k fully contains the set of its indices in W. (That is, every f_k is self-referential in W.) However, indices are contained not in the full range of f_k, but only in the intervals defined by the functions h and m starting from the k-th interval.

The class

$$U = \{f_n \mid n \in N\} \tag{7}$$

is not constructively identifiable in W. First of all, W is not recursive due to non-recursiveness of h and m. That is to say, all the information an IIM can rely on is the self-referential values of f_k. If there existed a value x_0 such that every f_k would output a self-reference given x_0, the problem of constructing an IIM would be trivial; however, every f_k does not output self-references up to the argument value from the k-th interval — that is, no self-referential interval is common for all the f_k. So the only possibility left for identifying U is to possess some "knowledge" about the structure of infinitely many such intervals; this is also not possible due to the incomputable properties of h and m stated above.

Nevertheless, U is reliably F-nonconstructively FIN-identifiable in W. For every f_k, we define the set

$$\pi(f_k) = \{h(i) \mid i \geq k\} \tag{8}$$

It is easy to see that an IIM defined as

$$M(\langle\langle f(0), f(1), ...\rangle, \pi_0 \in \pi(f)\rangle) = \langle f(\pi_0), f(\pi_0), ...\rangle \tag{9}$$

F-nonconstructively FIN-identifies U. Moreover, this identification is E-reliable with $E(x) \equiv \lfloor \frac{m(x)}{2} \rfloor$. Indeed, the condition (2) for E does hold, while any additional information word (8), even having been biased by E, would still help FIN-identify U.

What is left for us is to prove the $NK\text{-}Ref\text{-}FIN$ part. For any input object u, define the infinite additional information word

$$\langle h(0), h(1), ...\rangle \tag{10}$$

(We assume that the elements are mutually separated using some meta symbols.)

The IIM waits until the element $h(i) = u(1)$. If $h(i) > u(1)$ or i is not prime (or is less than 3), IIM outputs "#" and stops; otherwise it continues running the following algorithm:

1. Set $x \leftarrow 2$;
2. Calculate $(p_i)^x$;
3. Wait until the element $h((p_i)^x)$;
   - **a.** If $h((p_i)^x) = u(x)$:
     - **i.** Output $u(h(p_i))$ (if it is already received);
     - **ii.** Set $x \leftarrow x + 1$;
     - **iii.** Go to Step 2;
   - **b.** If $h((p_i)^x) \neq u(x)$:
     - **i.** Output "#";
     - **ii.** Stop execution.

It is quite obvious that such an IIM $NK\text{-}Ref\text{-}FIN$-identifies U with infinite nonconstructivity. ■

6 Conclusions and future work

We have shown that the classes $NK\text{-}R\text{-}X$ and $R\text{-}NK\text{-}X$ are not empty; moreover, the intersection of these classes is not empty. However, the above construction is not strong enough to allow proving (or disproving) the strongest case $R\text{-}NK\text{-}R\text{-}X \neq \varnothing$. Indeed, the refutable part of the proof relies on exact comparison of the additional information and the object in question, — that is, no errors in the given help could be allowed.

On the other hand, in the current paper we did not distinguish between e.g. $Rel\text{-}NK\text{-}X$ and $Ref\text{-}NK\text{-}X$. Moreover, only the F-type nonconstructivity was studied, while the other nonconstructivity types could be considered for reliable/refutable identification as well. We hope to give a more complete hierarchy of reliably nonconstructive identification classes in the future.

7 Acknowledgment

The authors thanks the anonymous referees of MEMICS 2010 for their valuable comments which improved the presentation of this paper.

References

1 A. Ambainis, K. Apsītis, C. Calude, R. Freivalds, M. Karpinski, T. Larfeldt, I. Sala, J. Smotrovs: "Effects of Kolmogorov Complexity Present in Inductive Inference as Well"; ALT '97: Proceedings of the 8th International Conference on Algorithmic Learning Theory (1997), 244-259.

2 L. Blum, M. Blum: "Toward a mathematical theory of inductive inference"; Information and Control, vol. 28, 1975, 125-155.

3 J. Case, C. Smith: "Comparison of Identification Criteria for Machine Inductive Inference"; Theoretical Computer Science 25 (1983), 193-220.

4 R. Freivald, R. Wiehagen: "Inductive inference with additional information"; Electron. Inform. Kybernetik 15, 1979, 179-195.

5 R. Freivalds, J. Bārzdiņš, K. Podnieks: "Inductive inference of recursive functions: complexity bounds"; Lecture Notes in Computer Science, vol. 502 (Baltic Computer Science) (1991), 111-155.

6 R. Freivalds: "Amount of nonconstructivity in finite automata": Lecture Notes in Computer Science, vol. 5642 (2009), 227-236.

7 M. Fulk: "Inductive Inference with Additional Information" (revised July 13, 1998 by S. Jain, A. Sharma): Journal of Computer and System Sciences 64 (2002), 153-159.

8 J. Gallier. "Formal Languages And Automata; Models of Computation, Computability; Basics of Recursion Function Theory"; `http://www.cis.upenn.edu/~jean/gbooks/tocnotes.html` (2010.04.14)

9 J. Gallier, A. Hicks: "The Theory of Languages and Computation"; `http://www.cis.upenn.edu/~jean/gbooks/tc.html` (2010.04.14)

10 E.M. Gold: "Language identification in the limit"; Information and Control 10 (1967), 447-474.

11 S. Jain, A. Sharma: "Learning in the Presence of Partial Explanations"; Information and Computation, vol. 95-2 (Dec 1991), 162-191.

12 S. Jain, A. Sharma: "Learning with the Knowledge of an Upper Bound on Program Size"; Information and Computation, vol. 102-1 (Jan 1993), 118-166.

13 I. Kucevalovs: "Nekonstruktivitātes daudzums induktīvajā izvedumā" (master thesis); University of Latvia, 2010. (in Latvian)

14 Lange, Zeugmann, Zilles: "Learning indexed families of recursive languages from positive data: a survey"; Theoretical Computer Science, vol. 397 (2008) n.1-3, 194-232.

15 M. Li, P. Vitányi: "An Introduction to Kolmogorov complexity and its applications" (3rd ed.); Springer-Verlag (2008).

16 E. Minicozzi: "Some natural properties of strong identification in inductive inference"; Theoretical Computer Science, vol.2, 1976, 345-360.

17 Y. Mukouchi, S. Arikawa: "Inductive Inference Machines That Can Refute Hypothesis Spaces"; tech. rep. RIFIS-TR-CS-67, Research Institute of Fundamental Information Science, Kyushu University, Jan. 15, 1994.

18 H. Rogers: "Theory of Recursive Functions and Effective Computability"; McGraw-Hill (1967).

19 A. Sakurai: "Inductive Inference of Formal Languages from Positive Data Enumerated Primitive-Recursively"; The Japan Society for Industrial and Applied Mathematics, Vol.1 No.3, 1991, pp.177-193. (in Japanese)

20 Zeugmann, Zilles: "Learning Recursive Functions: A Survey"; Theoretical Computer Science, vol. 397 (2008) n.1-3, 4-56.

Simultaneous Tracking of Multiple Objects Using Fast Level Set-Like Algorithm

Martin Maška, Pavel Matula, and Michal Kozubek

Centre for Biomedical Image Analysis, Faculty of Informatics
Masaryk University, Brno, Czech Republic
xmaska@fi.muni.cz

Abstract

A topological flexibility of implicit active contours is of great benefit, since it allows simultaneous detection of several objects without any a priori knowledge about their number and shapes. However, in tracking applications it is often required to keep desired objects mutually separated as well as allow each object to evolve itself, i.e., different objects cannot be merged together, but each object can split into several regions that can be merged again later in time. The former can be achieved by applying topology-preserving constraints exploiting either various repelling forces or the simple point concept from digital geometry, which brings, however, an indispensable increase in the execution time and also prevent the latter. In this paper, we propose more efficient and more flexible topology-preserving constraint based on a region indication function, that can be easily integrated into a fast level set-like algorithm [15] in order to obtain a fast and robust algorithm for simultaneous tracking of multiple objects. The potential of the modified algorithm is demonstrated on both synthetic and real image data.

Keywords and phrases level set framework, topology preservation, object tracking

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.69

1 Introduction

Detection and tracking of object boundaries is an important task in many computer vision applications such as video surveillance, monitoring, or robotics as well as in biomedical studies aimed at understanding the mechanics of cellular processes such as proliferation, differentiation, or migration. In general, desired objects can have arbitrary initial shapes that can, in addition, undergo changes in time. Therefore, an optimal tracking algorithm should be able to detect objects of complex boundaries and adapt easily to their changes. Furthermore, it should also achieve real-time or at least near real-time performance in order to be fruitfully applied in practice.

Implicit active contours [4, 5, 6, 23] have become popular namely due to their inherent topological flexibility and ability to detect objects of complex shapes. Their solution is usually carried out using the level set framework [19, 18], in which the contour is represented implicitly as the zero level set (also called *interface*) of a scalar higher-dimensional function. This representation has several advantages over the parametric one [10, 3]. In particular, it avoids parameterization problems, the topology of the contour is handled inherently, and the extension into higher dimensions is straightforward. On the other hand, a numerical solution of associated partial differential equations brings a significant computational burden limiting the use of this approach in real-time applications.

Many approximations, aimed at speeding up the basic level set framework, have been proposed in last two decades. In the family of gradient-based implicit active contours [4, 5], the narrow band [1], sparse-field [25], and fast marching method [20] have become popular. Later, other interesting approaches based on the additive operator splitting scheme [8]

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 69–76

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

or a pointwise scheduled propagation of the implicit contour [7, 17] have emerged. Shi and Carl [22] proposed a fast algorithm that is able to track the gradient-based as well as region-based [6, 23] implicit active contours, provided the speed function can be decomposed into data-dependent and regularization terms. We also refer the reader to the work by Lie et al. [12], Wang et al. [24], and Maška et al. [15] introducing other fast algorithms that minimize popular Chan-Vese model [6].

The topological flexibility of implicit active contours is of great benefit, since it enables to detect several objects simultaneously without any a priori knowledge about their number or shapes. However, for tracking purposes such a flexibility is not always suitable. For instance, when two initially isolated objects touch later in time it is often required to keep them separated. This can be achieved by applying topology-preserving constraints based on either various repelling forces [2, 11] or the simple point concept from digital geometry [9, 14], which brings, however, an indispensable increase in the execution time caused by their evaluation in a local neighbourhood of the interface. Furthermore, they also prevent each object from being evolved at will, e.g., from splitting into several regions.

In this paper, we propose more flexible topology-preserving constraint that brings only negligible increase in the execution time. It exploits a region indication function, has constant time complexity, and can be easily integrated into our fast level set-like algorithm [15] in order to obtain a fast and robust algorithm for simultaneous tracking of multiple objects based on the minimization of the Chan-Vese model [6]. In comparison to the tracking algorithm by Shi and Carl [21] that exploits the region indication function as well, the proposed algorithm does not require the contours to be initially separated by the background nor evaluate relaxed topological numbers. It also allows two different object contours to touch inherently, without any additional tests.

The organization of the paper is as follows. In Section 2, the theoretical background of the Chan-Vese model and the basic principle of our fast level set-like algorithm [15] intended for its minimization are reviewed. Section 3 is devoted to the topology-preserving modification of the original algorithm. Experimental results are demonstrated in Section 4. We conclude the paper with a discussion and suggestions for future work in Section 5 and 6, respectively.

2 Fast Algorithm Minimizing the Chan-Vese Model

In order to obtain a mathematically easier minimization problem, Chan and Vese [6] introduced a piecewise constant approximation to the well-known functional formulation of image segmentation by Mumford and Shah [16]. Let Ω be an image domain and $u_0 : \Omega \to \mathbb{R}$ be an input image defined over this domain. The basic idea of the Chan-Vese model is to find a piecewise constant approximation of u_0 being constant in two possibly disconnected regions Ω_1 and Ω_2 of constant levels c_1 and c_2, respectively, separated by a closed segmenting contour C ($\Omega = \Omega_1 \cup \Omega_2 \cup C$) of minimal length. The Chan-Vese model can be formulated as

$$E_{CV}(C, c_1, c_2) = \mu|C| + \lambda_1 \int_{\Omega_1} (u_0(x) - c_1)^2 \, dx + \lambda_2 \int_{\Omega_2} (u_0(x) - c_2)^2 \, dx \ , \tag{1}$$

where μ is nonnegative and λ_1 and λ_2 are positive constants. Embedding the contour C in a scalar higher-dimensional function ϕ with C as its zero level set, the functional can be minimized using the level set framework. The associated Euler-Lagrange equation has the following form:

$$\frac{\partial \phi}{\partial t} + \delta_\varepsilon(\phi) \left[\mu \cdot \operatorname{div} \left(\frac{\nabla \phi}{|\nabla \phi|} \right) - \lambda_1 (u_0 - c_1)^2 + \lambda_2 (u_0 - c_2)^2 \right] = 0 \ , \tag{2}$$

where

$$c_1 = \frac{\int_\Omega u_0(x)(1 - H_\varepsilon(\phi(x)))\,dx}{\int_\Omega (1 - H_\varepsilon(\phi(x)))\,dx} \quad \text{and} \quad c_2 = \frac{\int_\Omega u_0(x) H_\varepsilon(\phi(x))\,dx}{\int_\Omega H_\varepsilon(\phi(x))\,dx} \ . \tag{3}$$

The symbols H_ε and δ_ε denote regularized versions of the Heaviside and Dirac delta functions. Careful attention has to be paid to the regularization of these functions, since it affects the model performance. Provided δ_ε is nonzero in the whole domain, the Chan-Vese model has the tendency to compute a global minimizer. On the contrary, the choice of δ_ε with a compact support results only in a local minimizer and, therefore, the dependence on the initialization.

In our previous work [15], we introduced a fast level set-like algorithm that locally minimizes the Chan-Vese model (a suitable choice of initial model, however, often leads to finding a global minimum) and avoids a nontrivial and time-consuming numerical solution of the associated Euler-Lagrange equation. Instead of evolving the whole implicit function in a small time step, only the interface points stored in a list data structure are moved to the exterior or interior depending on the sign of the speed function F in the normal direction given as

$$F = \mu\kappa - \lambda_1(u_0 - c_1)^2 + \lambda_2(u_0 - c_2)^2 \ , \tag{4}$$

where κ denotes the curvature of the interface. Simultaneously, their local neighbourhoods (4-neighbourhoods in 2D and 6-neighbourhoods in 3D, respectively) are updated accordingly. The local propagation of each interface point allows the values c_1 and c_2 to be updated incrementally, since we know exactly which points move to the exterior and interior. Furthermore, considering the level set function ϕ as a mapping of the set membership of each point (i.e. the points of the interface are represented by the value 0, interior points by -1, and exterior ones by 1), the curvature of the interface can be roughly approximated in an incremental manner. These ideas result in a fast algorithm for tracking implicit contours driven by the Chan-Vese model. We refer the reader to the original paper [15] for further details.

3 Topology-Preserving Modification

To ensure that different objects are kept mutually separated as well as allow each object to evolve itself, we integrate our fast algorithm described in the previous section with a region indication function $\psi : \Omega \to \{0, 1, 2, \dots\}$ that is evolved simultaneously with the simplified level set function ϕ. Remind that in each iteration the original algorithm propagates each interface point locally depending on the sign of the speed function F. Therefore, a modification of the local propagation of each interface point will result in a modification of the original algorithm itself.

Let ϕ be determined by a possibly disconnected background region Ψ_0 and M possibly disconnected disjoint objects $\Psi_1, \Psi_2, \dots, \Psi_M$ ($\Omega = \bigcup_{0 \le i \le M} \Psi_i$). Let $p \in \Omega$ be a point of the interface of the object Ψ_i, $0 < i \le M$, that is being propagated. The behaviour of the modified algorithm can be divided into two cases depending on the sign of $F(p)$. First, assume that $F(p) < 0$ (Fig. 1a). The original algorithm transfers p to the exterior and adds all its interior neighbours to the interface. The modified algorithm behaves in the same way as the original one. Clearly, only p is switched from the foreground to the background. It is therefore sufficient to reset its region indicator to 0.

The second case, when $F(p) > 0$ (Fig. 1b), is more complicated than the first one. The original algorithm transfers p to the interior and adds all its exterior neighbours (denote them

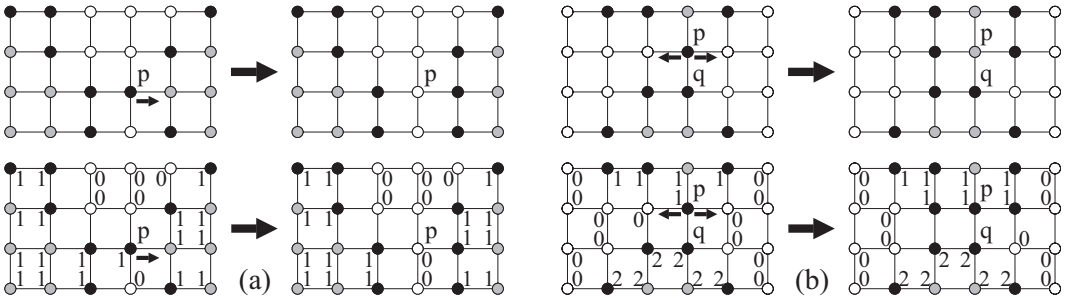


Figure 1 Comparison of one iteration of the original algorithm (top row) and the modified one (bottom row) in case of (a) $F(p) < 0$ and (b) $F(p) > 0$. The black points correspond to the interface, the white ones to the exterior, and the gray ones to the interior. The arrows from p correspond to the directions of possible propagations of the interface in this iteration. The numbers correspond to the region indication function ψ.

by $E(p)$) to the interface. In this case, each point in $E(p)$ is switched from the background to the foreground. Therefore, the modified algorithm changes their region indicators to i. It is important to note that one more test has to be performed in the modified algorithm in order to preserve the interface connectedness of each object. Let $N(p)$ be a set of neighbours of p of different region indicators. Clearly, if $|E(p)| < |N(p)|$, p must be put back to the interface, $\phi(p) = 0$, in order to preserve the interface connectedness of the object Ψ_i, since p has a neighbour q of the region indicator j, $0 < j \leq M$, $j \neq i$, that belongs to the interface of the object Ψ_j.

4 Experimental Results

In this section, we present several results and comparisons on both synthetic and real image data to demonstrate the potential of the proposed algorithm. The experiments have been performed on a common workstation (Intel Core2 Duo 2.0 GHz, 2 GB RAM, Windows XP Professional). For comparison purposes, we integrated the original algorithm [15] with the simple point concept from digital geometry to obtain a fast topology-preserving alternative to the modified algorithm described in the previous section. We denote these algorithms as SP (simple point) and RI (region indicator), respectively, depending on the concept used for preserving the contour topology.

We start with a synthetic binary image of size 200×200 pixels containing two circles (Fig. 2). In case of the SP algorithm, the contour cannot change its topology and, therefore, only one 8-connected component is obtained as a result. On the other hand, the RI algorithm allows the contour to split into several parts and each circle is detected separately. The execution time was less than 0.01 seconds in both cases.

The second experiment is aimed at separation of two touching objects in a noisy synthetic image of size 350×170 pixels (Fig. 3). Both algorithms output two 8-connected components. However, in case of the SP algorithm they are separated by often undesired 4-connected background path. The computation took 0.014 and 0.013 seconds, respectively.

We conclude this section with an application of the SP and RI algorithms for tracking of AIF-transfected living cells of the MCF-7 cell line (Fig. 4 and 5, respectively). The time-lapse series acquired using a fluorescence microscope has 25 frames of size 648×515 pixels. The execution time was about 0.111 and 0.107 seconds, respectively, in average per frame.

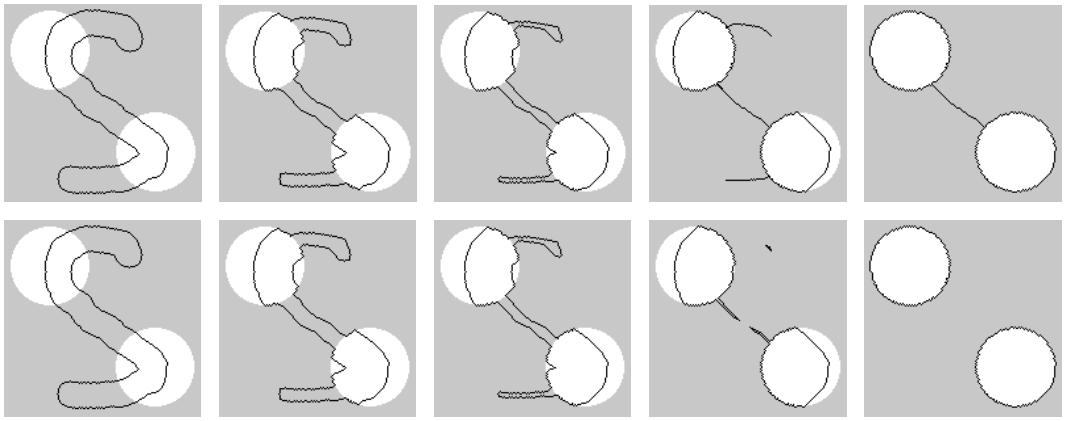


Figure 2 Segmentation of a synthetic image with two circles ($\mu = 0.5$, $\lambda_1 = \lambda_2 = 1$). Top row: Evolution of the SP contour. Bottom row: Evolution of the RI contour.

Figure 3 Segmentation of touching objects ($\mu = 0.5$, $\lambda_1 = \lambda_2 = 1$). Left: Input image overlaid with two initial contours. Centre: Segmentation result of the SP algorithm. Right: Segmentation result of the RI algorithm.

5 Discussion

The final evaluation of the modified algorithm is introduced in this section. We discuss, namely, the experimental results presented in Sect. 4 in detail.

The topology-preserving constraint exploiting the region indication function is very simple and has constant time complexity. There is no need to evaluate any complex condition in a local neighbourhood of a considered point. In comparison to the original algorithm, the increase in the execution time of the modified algorithm is negligible, from about 2 up to 4 percent in both 2D as well as 3D. Compared with the SP algorithm, it is about 4 percent faster in 2D and even about 9 percent faster in 3D, where the breadth-first search algorithm [13] has been used for the simple point detection. On the other hand, the RI algorithm consumes slightly more memory than the others, since it requires additional space for storing region indicators. However, the increase is less than 5 percent.

The experiments illustrated in Fig. 2–5 showed the main advantages of the RI algorithm over the SP one for simultaneous tracking of multiple objects. Considering the simplest tracking scheme in which the final contour from the previous frame is used as a seed in the next one, the RI algorithm adapts easily to splitting of a connected object in one frame into several regions in the next one. Furthermore, it also allows us to find boundaries of touching objects without any background gap between them. It is important to note that considered tracking scheme might have problems in situations involving large movements of the objects or when the final contour of one object from the previous frame overlaps with another object in the next frame. This will be addressed in future work.

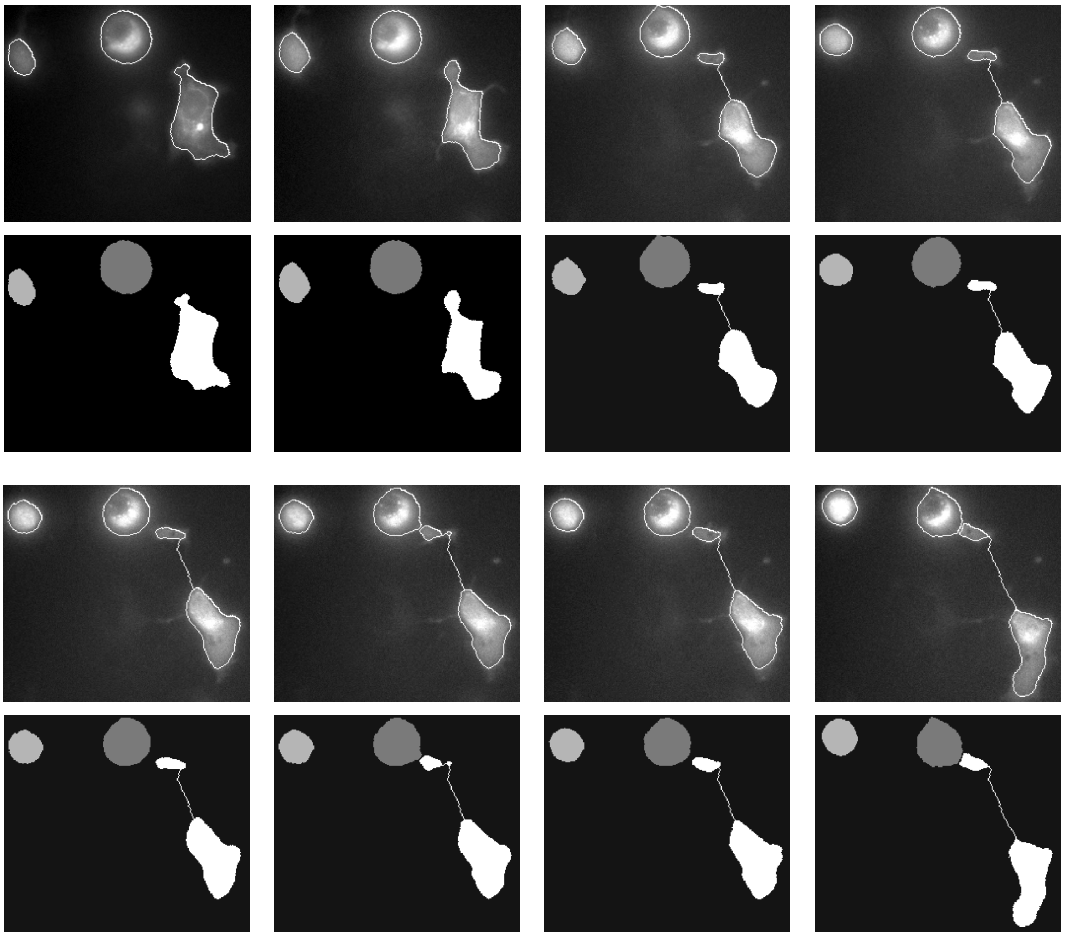


Figure 4 Tracking of AIF-transfected living cells using the SP algorithm ($\mu = 0.3$, $\lambda_1 = 1$, $\lambda_2 = 2$). The frames number 1, 3, 7, 10, 13, 14, 15, and 22 are shown. Top rows: Original image data overlaid with final contours. Bottom rows: Segmentation results of the SP algorithm.

6 Conclusion

We have addressed the problem of imposing topology-preserving constraints on evolving implicit contours. We have proposed a topology-preserving constraint exploiting a region indication function, that is more flexible than the ones based on either various repelling forces or the simple point concept from digital geometry, has constant time complexity, and can be easily integrated into our fast level set-like algorithm minimizing the Chan-Vese model. The experiments verified topology-preserving properties of the modified algorithm and showed its speed and better usability for simultaneous tracking of multiple objects in comparison to the one exploiting the simple point concept.

Acknowledgments. This work has been supported by the Ministry of Education of the Czech Republic (Projects No. MSM-0021622419, No. LC535 and No. 2B06052). The authors would also like to thank Dr. Miroslav Vařecha for providing the time-lapse series of MCF-7 cell line.

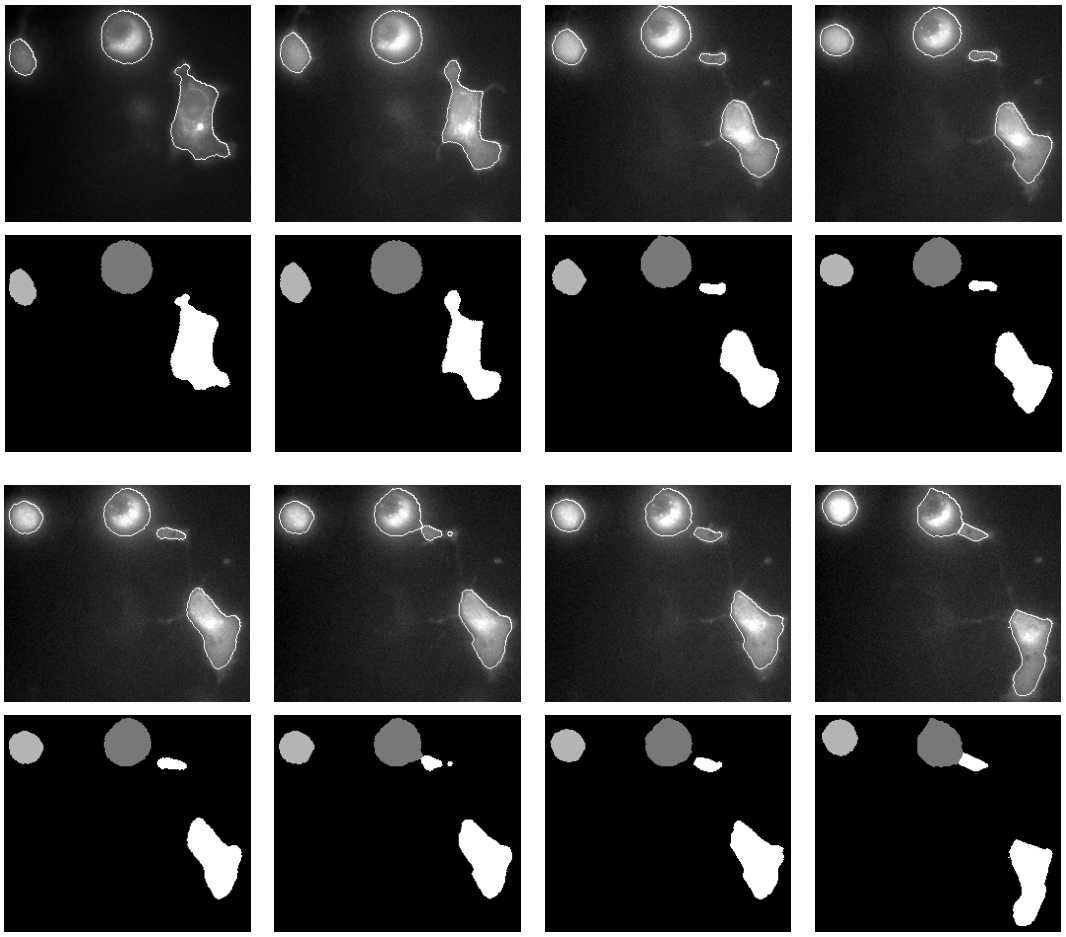


Figure 5 Tracking of AIF-transfected living cells using the RI algorithm ($\mu = 0.3$, $\lambda_1 = 1$, $\lambda_2 = 2$). The frames number 1, 3, 7, 10, 13, 14, 15, and 22 are shown. Top rows: Original image data overlaid with final contours. Bottom rows: Segmentation results of the RI algorithm.

References

1 D. Adalsteinsson and J. A. Sethian. A fast level set method for propagating interfaces. *Journal of Computational Physics*, 118(2):269–277, 1995.

2 O. Alexandrov and F. Santosa. A topology-preserving level set method for shape optimization. *Journal of Computational Physics*, 204(1):121–130, 2005.

3 P. Brigger, J. Hoeg, and M. Unser. B-spline snakes: A flexible tool for parametric contour detection. *IEEE Transactions on Image Processing*, 9(9):1484–1496, 2000.

4 V. Caselles, F. Catté, T. Coll, and F. Dibos. A geometric model for active contours in image processing. *Numerische Mathematik*, 66(1):1–31, 1993.

5 V. Caselles, R. Kimmel, and G. Sapiro. Geodesic active contours. *International Journal of Computer Vision*, 22(1):61–79, 1997.

6 T. F. Chan and L. A. Vese. Active contours without edges. *IEEE Transactions on Image Processing*, 10(2):266–277, 2001.

7 J. Deng and H. T. Tsui. A fast level set method for segmentation of low contrast noisy biomedical images. *Pattern Recognition Letters*, 23(1-3):161–169, 2002.

8 R. Goldenberg, R. Kimmel, E. Rivlin, and M. Rudzsky. Fast geodesic active contours. *IEEE Transactions on Image Processing*, 10(10):1467–1475, 2001.

9 X. Han, C. Xu, and J. L. Prince. A topology preserving level set method for geometric deformable models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(6):755–768, 2003.

10 M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–331, 1987.

11 C. Le Guyader and L. A. Vese. Self-repelling snakes for topology-preserving segmentation models. *IEEE Transactions on Image Processing*, 17(5):767–779, 2008.

12 J. Lie, M. Lysaker, and X. C. Tai. A binary level set model and some applications to Mumford-Shah image segmentation. *IEEE Transactions on Image Processing*, 15(5):1171–1181, 2006.

13 G. Malandain and G. Bertrand. Fast characterization of 3d simple points. In *Proceedings of 11th International Conference on Pattern Recognition*, pages 232–235, 1992.

14 M. Maška and P. Matula. A fast level set-like algorithm with topology preserving constraint. In *Proceedings of the 13th International Conference on Computer Analysis of Images and Patterns*, LNCS 5702, pages 930–938. Springer-Verlag, 2009.

15 M. Maška, P. Matula, O. Daněk, and M. Kozubek. A fast level set-like algorithm for region-based active contours. In *Proceedings of the 6th International Symposium on Visual Computing*, LNCS 6455, pages 387–396. Springer-Verlag, 2010.

16 D. Mumford and J. Shah. Optimal approximation by piecewise smooth functions and associated variational problems. *Communications on Pure and Applied Mathematics*, 42(5):577–685, 1989.

17 B. Nilsson and A. Heyden. A fast algorithm for level set-like active contours. *Pattern Recognition Letters*, 24(9-10):1331–1337, 2003.

18 S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer-Verlag New York, Inc., Secaucus, USA, 2003.

19 S. Osher and J. A. Sethian. Fronts propagating with curvature dependent speed: Algorithms based on Hamilton–Jacobi formulation. *Journal of Computational Physics*, 79(1):12–49, 1988.

20 J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

21 Y. Shi and W. C. Carl. Real-time tracking using level sets. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 34–41, 2005.

22 Y. Shi and W. C. Karl. A real-time algorithm for the approximation of level-set-based curve evolution. *IEEE Transactions on Image Processing*, 17(5):645–656, 2008.

23 A. Tsai, A. Yezzi, and A. S. Willsky. Curve evolution implementation of the Mumford-Shah functional for image segmentation, denoising, interpolation, and magnification. *IEEE Transactions on Image Processing*, 10(8):1169–1186, 2001.

24 X. F. Wang, D. S. Huang, and H. Xu. An efficient local chan-vese model for image segmentation. *Pattern Recognition*, 43(3):603–618, 2010.

25 R. T. Whitaker. A level-set approach to 3D reconstruction from range data. *International Journal of Computer Vision*, 29(3):203–231, 1998.

GPU-Based Sample-Parallel Context Modeling for EBCOT in JPEG2000*

Jiří Matela[1,3], Vít Rusňák[1], and Petr Holub[2,3]

1 **Faculty of Informatics, Masaryk University**
Botanická 68a, 602 00 Brno, Czech Republic
matela@ics.muni.cz, xrusnak@fi.muni.cz
2 **Institute of Computer Science, Masaryk University**
Botanická 68a, 602 00 Brno, Czech Republic
hopet@ics.muni.cz
3 **CESNET z.s.p.o.**
Zíkova 4, 162 00 Prague, Czech Republic

Abstract

Embedded Block Coding with Optimal Truncation (EBCOT) is the fundamental and computationally very demanding part of the compression process of JPEG2000 image compression standard. EBCOT itself consists of two tiers. In Tier-1, image samples are compressed using context modeling and arithmetic coding. Resulting bit-stream is further formated and truncated in Tier-2. JPEG2000 has a number of applications in various fields where the processing speed and/or latency is a crucial attribute and the main limitation with state of the art implementations. In this paper we propose a new parallel approach to EBCOT context modeling that truly exploits massively parallel capabilities of modern GPUs and enables concurrent processing of individual image samples. Performance evaluation of our prototype shows speedup 12 times for the context modeller, and 1.4–5.3 times for the whole EBCOT Tier-1, which includes not yet optimized arithmetic coder.

Keywords and phrases JPEG2000, EBCOT, Context Modeling, GPU, GPGPU, parallel

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.77

1 Introduction

JPEG2000 [10] is an image compression standard created by the Joint Photographic Experts Group (JPEG). JPEG2000 is aimed at providing not only compression performance superior to the current JPEG standard but also advanced capabilities demanded by applications in the fields such as medical imaging [18], film industry [12], or image archiving. It features optional mathematically lossless processing, error resilience, or progressive image transmission by improving pixel accuracy and resolution. On the other hand, the advanced features and the superb compression performance yields higher computational demands which implies slower processing.

Graphics processing units (GPUs) have become a popular computing architecture in last half of decade due to their rapid increase of performance compared to traditional CPUs [16]. While parallel and hierarchical architecture of GPUs allows for impressive increase of performance at moderate cost, it requires specific regards when designing and implementing algorithms to utilize potential of the GPU (Section 2.2). Since JPEG2000

* This project has been supported by a research intents MŠM 0021622419 and MŠM 6383917201 and GD 102/09/H042 grant.

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 77–84

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

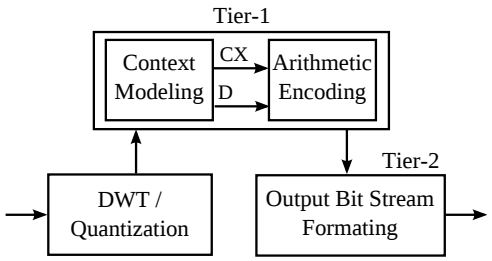


Figure 1 Block diagram of the JPEG2000 encoder

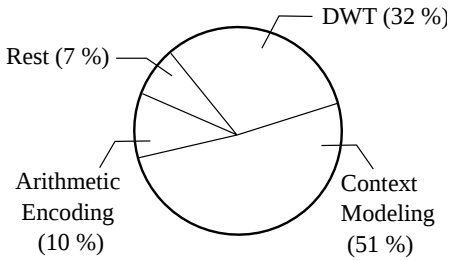


Figure 2 Profiling status of OpenJPEG implementation using benchmarked HD image on testbed described in Section 5.

introduction, there has been a great deal of effort to provide JPEG2000 applications with sufficient processing speed and bandwidth. The majority of this effort has its base in FPGA and VLSI in general [3, 11, 20]. As for the GPU computing, there has been attempts [17, 19] to coarse-grained parallelization resulting in performance very close to CPU implementations. Our goal is adaptation or re-formulation of individual algorithms resulting in fine-grained and more effective design which fits the specifics of modern GPUs better.

The simplified block diagram of compression system defined by JPEG2000 standard is illustrated in Fig. 1. Prior to actual compression the image data is transformed using Discrete Wavelet Transform [6, 7, 14] (DWT). JPEG2000 standard prescribes use of CDF 9/7 and CDF 5/3 wavelet transform [5] for lossy and lossless compression modes respectively. In case of lossy compression, the transformed coefficients are quantized using uniform scalar dead-zone quantization [13]. The process of quantization introduces the data precision reduction in order to make it more compressible. Thereafter the data is compressed in EBCOT Tier-1 and the resulting bit-stream is further formated in Tier-2. As can be seen in Fig. 2, the most computationally intensive parts of JPEG2000 are DWT, Context Modeling, and Arithmetic Encoding.

This paper describes a novel fine-grained GPU-based parallel design of the context modeling part of JPEG2000. Section 2 provides background on context modeling in JPEG2000 and mentions GPU basics needed for further explanations of our design introduced in Section 4. Section 3 reviews related work. The evaluation methodology, experimental results and their discussion is in Section 5. Section 6 summarizes the key findings and presents directions for future work.

2 Preliminaries

As noted above, EBCOT is a two-tiered coder. The input to Tier-1 is DWT-transformed image partitioned into so called *code-blocks*[1]. Each code-block is processed independently in Tier-1 using context modeling and arithmetic coding to form an *embedded bit-stream* representing the compressed code-block. The context modeller analyzes the bit structure of the images and collects contextual information (CX) which is passed together with bit values (D) to the arithmetic coder. The JPEG2000 uses MQ-Coder—a context adaptive binary arithmetic coder—defined in JBIG2 standard [9]. The MQ-Coder codes bit values based on its context information. There is 19 different contexts defined and for each of

[1] Recommended code-block dimensions are 16×16, 32×32, and 64×64. The total number of code-block samples may not exceed 4096.

them, the arithmetic coder maintains and consecutively adapts probability estimate [4, 2]. Final compressed bit-stream is formatted during Tier-2, where the embedded bit-streams are combined so that the desired rate-distortion criteria is fulfilled.

The following explanation of JPEG2000 and EBCOT processes uses only single color component of the image for sake of simplicity. This approach is possible because EBCOT Tier-1 processes color components independently [1, Chapter 6.6].

2.1 EBCOT Tier-1 Context Modeling

The context modeling module processes code-blocks bit-plane by bit-plane[2] starting from the most significant bit-plane (MSB). Each bit-plane is coded in three passes but each bit is processed in exactly one pass—i.e., each pass scans through the entire bit-plane but processes only some of the bits. The decision whether to process a bit in current pass or not is made based on current state of the bit and states of its neighbours. Note that the bit state information changes as the bits are processed; therefore, the process is defined sequentially with the prescribed scanning order to create and maintain correct state. The scanning order in the bit-plane is illustrated in [1, p. 166]. The three passes are *i)* Signification Propagation Pass (SPP), *ii)* Magnitude Refinement Pass (MRP), and *iii)* Clean-Up Pass (CUP). Each pass encodes a bit using one or more of the following four bit-coding operations defined by JPEG2000 standard: Zero Coding (ZC), Run-Length Coding (RLC), Magnitude Refinement Coding (MRC), and Sign Coding (SC). Based on bit values and state informations, these four operations generate 1–4 CX,D pairs per each bit in a bit-plane as input for the arithmetic encoder.

The state information consists of three state variables σ, σ', η. The σ and σ' states are shared by all the bits of a pixel, indicating that the first non-zero bit of the pixel has already been processed and that MRC coding has been applied, respectively. The η is not shared, and indicates the bit has been processed in SPP pass on the current bit-plane [1].

A bit is in a so called *preferred neighborhood* (PN) if at least one of its 8 adjacent neighbours is *significant*, i.e., has $\sigma = 1$. All bits having $\sigma = 0$ and being in the PN are coded in SPP pass. The bits of the pixels that have become significant in the previous bit-planes, are coded in second, MRP, pass. Those bits have $\sigma = 1$ and $\eta = 0$. The rest of bits in current bit-plane is processed in CUP pass—i.e., all bits having $\sigma = \eta = 0$ after the previous two passes.

2.2 GPU architecture and programming model

Attracted by their raw computing power, a number of general-purpose GPU computing approaches has been implemented in recent years, including GLSL[3], CUDA, and OpenCL[4]. Because of its flexibility and potential to utilize power of GPU, we have opted for CUDA (Compute Unified Device Architecture) [15]—a massively parallel computing architecture designed by Nvidia. In general, modern GPU architectures are, capable of running thousands of threads in parallel. In the context of CUDA, threads are grouped into so called thread blocks. Threads within the block can cooperate among themselves using synchronization

[2] Bit-plane is defined as one-bit image composed of the same bit of each pixel, see [8, Chapter 3]. Number of bit-planes corresponds to the number of bits per pixel for each color component of the image. Given the preceding DWT transformation, each "pixel" in actually a DWT coefficient generated by the transformation.

[3] `http://www.opengl.org/documentation/glsl/`

[4] `http://www.khronos.org/opencl/`

primitives, shared memory, and global memory. Compared to the global memory, the shared memory is considerably smaller and significantly faster and should be used whenever possible. The advantage of the global memory is that it can be accessed by all threads, whereas the shared memory is only visible to threads of one block. The common CUDA work flow is to copy data from RAM to the global memory of the GPU. All GPU threads can access and process the data directly in global memory, or, more preferably, the data can be partitioned and fetched into the shared memory to provide higher throughput for more complex operations. It is also important that threads within the same warp follow the same execution path; otherwise the thread divergence is introduced and divergent execution paths are serialized, thus worsening performance.

3 Related Work

JPEG2000 standard allows for code-block level parallelism, which is rather coarse-grained and because of intermediate data size requirements, it enforces use of global memory on CUDA platform. Another option is stripe-level parallelism in casual mode, which has lesser requirements on memory but results in worse compression performance. Sequential nature of the context modeller requires processing of one code-block/stripe by a single thread only; thus yielding (a) not enough threads too utilize massively parallel architecture of GPUs and (b) code divergence that introduces further performance penalty.The code-block level parallelism has been used by the CUJ2K [19], an open source JPEG2000 project which uses CUDA architecture and its programming model to implement all compute intensive parts for GPU. A design similar to CUJ2K has been proposed by Datla et al. in [17].

4 Context Modeling Parallelization for GPU Architectures

Compared to the coarse-grained parallelism contained within JPEG2000 standard, the bit-parallel context modeling architecture proposed by us allows for independent processing of all samples of a bit-plane as well as independent processing of all bit-planes. Our design bypasses the three coding passes (SPP, MRP, CUP) and the prescribed scan pattern, enabling direct coding by the four bit-coding operations (ZC, MRC, RLC, SC).

For the purposes of the following explanation we define a code-block as two-dimensional sequence of samples, $\gamma_{x,y}$ $(x = 1..m, y = 1..n)$, m and n being the horizontal and vertical code-block dimensions respectively. A binary representation of a sample γ is a sequence $[\gamma^{P-1}, \gamma^{P-2}, \ldots, \gamma^1, \gamma^0]$ where P is image bit depth. $\gamma^p_{x,y}$ thus denotes a bit of the sample $[x, y]$ on bit-plane p.

To be able to bypass the passes and to enable the direct coding, we introduce two new state variables $\rho^p_{x,y}$, and $\tau^p_{x,y}$ as replacement to the original states. The meaning of the two new state variables is as follows: $\rho^p_{x,y}$ is shared by all the bits of each pixel and $\rho^p_{x,y} = 1$ indicates the pixel $\gamma_{x,y}$ became significant in either p or in one of the previous bit-planes according to the processing order; $\tau^p_{x,y} = 1$ indicates $\gamma_{x,y}$ is going to become significant during SPP on the current bit-plane.

To be able to code a bit-plane p in parallel, the two new coding states need to be precomputed before the actual coding. The $\rho^{p+1}_{x,y}$ is computed in parallel by examining the previous $p + 1$ bit-planes; $\rho^{p+1}_{x,y} = 1$ iff there is a non-zero bit above current bit, i.e. $\bigvee_{p'=p+1}^{P} \gamma^{p'}_{x,y} = 1$.

The $\tau^p_{x,y}$ is inductively computed in parallel as follows:

- $\tau^p_{x,y} = 1\ \forall[x,y]$ where $\rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge$ at least one of 8 adjacent neighbors has $\rho^{p+1} = 1$
- In each further step $\tau^p_{x,y} = 1\ \forall[x,y]$ where $\rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge$ (at least one of 8 adjacent neighbors has $\rho^{p+1} = 1 \vee$ one of four preceding neighbours[5] has non-zero τ^p).

	Original	**New**
MRC	$\sigma_{x,y} = 1 \wedge \eta_{x,y} = 0$	$\rho^{p+1}_{x,y} = 1$
RLC	$\sigma_{x,y} = 1 \wedge \eta_{x,y} = 0 \wedge y$ is a multiple of $4 \wedge \sum_{i=x-1}^{x+1} \sum_{j=y-1}^{y+4} \sigma_{i,j} = 0$	$\rho^{p+1}_{x,y} = 0 \wedge$ is in PN $\wedge$ $\left(\sum_{i=x-1}^{x+1} \sum_{j=y-1}^{y+4} (\rho^{p+1}_{i,j} + \tau^p_{i,j}) + \sum_{j=y-1}^{y+3} \gamma^p_{x-1,j} + \gamma^p_{x,y-1} = 0\right)$
ZC	$\sigma_{x,y} = 0 \wedge$ [in PN (for SPP) or $\eta_{x,y} = 0$ (for CUP)]	$\rho^{p+1}_{x,y} = 0$ (PN differentiate SPP from CUP)
SC	(SPP or CUP preconditions) $\wedge\ \gamma^p_{x,y} = 1$	$\rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1$ ($\tau^p_{x,y}$ differentiates SPP from CUP)

Table 1 Overview of preconditions of coding operations.

Once both ρ and τ state variables are computed, the coding operations for an arbitrary bit $\gamma^p_{x,y}$ can be decided. In order to avoid execution path divergence on GPU, we propose to serialize the coding operations execution manually and to implement bit-to-thread mapping—i.e., the thread-blocks are of the same dimension as the code-blocks; each bit-plane is processed in the following four consecutive steps: MRC, RLC, ZC, SC. Note, that each coding operation is executed on a bit-plane in parallel. The only constraint on bit coding independence stems from diverging number of bits coded by the RLC operation. The RLC is defined to code one to four bits in column and a prediction of the number is virtually as expensive as the RLC coding itself. The only operation affected by this is ZC, so we choose to perform RLC operations on current bit plane before ZC. Although the new design we propose allows for parallelism among bit-planes too, we do not exploit it because of restricted shared memory size. Direct selection of coding operations based on the new state variables compared to the original sequential state variables is summarized in Table 1. A detailed equivalence proof is beyond the size limitation of this paper.

State information is also needed by the coding operations. To code the bits, the original coding operations use σ, SC also exploits pixel sign information, and MRC uses σ' state. The new state variables are used instead as follows:

- MRC uses ρ^{p+1} and τ^p of all the neighbors instead of σ; the σ' is substituted by looking for the position of the first non-zero bit on previous $p+1$ bit-planes.
- instead of σ, ZC uses ρ^{p+1} of all neighbors and τ^p of four preceding neighbors for bits belonging to SPP. ρ^{p+1} and τ^p all neighbors and bit value of the four preceding neighbors are used for bits belonging to CUP.
- instead of σ, SC uses ρ^{p+1} of two vertical and two horizontal neighbors and τ^p of the upper and the left side neighbor for bits belonging to SPP. ρ^{p+1} and τ^p of two vertical and two horizontal neighbors and bit value of of the upper and the left side neighbor for bits belonging to CUP.
- RLC uses no state information at all, both prior and after the transformation.

[5] The four preceding neighbors of $[x,y]$ are as follows: $[x,y-1]$; $[x-1,y-1]$; $[x-1,y]$; $[x-1,y+1]$.

The described fine-grained parallel algorithm allows for processing individual bits in parallel threads, resulting in high utilization of multi-processors on GPU. Depending on chosen code block size, the data may be processed entirely in the fast shared memory[6].

5 Experimental Evaluation

5.1 Methodology

We implemented two benchmark sets focused on the EBCOT Tier-1 processing speed of selected single-threaded CPU implementations (OpenJPEG[7], JasPer[8] and Kakadu[9]) and GPU implementation (CUJ2K[10]) together with our GPU implementation nicknamed bpcuda. Except for Kakadu, all the implementations are open-source—this allowed us to add additional timer functions to the source codes to obtain comparable results. Kakadu codec introduced two limitations: (a) only the timer provided by the Kakadu authors could be used, (b) the benchmarking of all the implementations comprises run-time of the whole EBCOT Tier-1, not just the context modeller, to make results directly comparable. Further insight into EBCOT Tier-1 components has been implemented using the best open-source CPU and GPU implementations: JasPer and bpcuda.

Primary input image parameters affecting processing speed are size and bit-depth. The image content itself also affects the runtime of EBCOT Tier-1; thus we selected two extreme cases and one standard image for the first benchmark set: a single-color image, a white-noise image, and *Lenna* image, a well-known picture which is broadly used for benchmarking purposes. All three images were 8-bit grayscale with the same size of 512×512 pixels. The second benchmark set was focused on dependency analysis of processing time on image size: three images with the same content (a real-world digital photography portrait) and different size have been used. Images were 8-bit grayscale with the size of 1280×720, 1920×1080 and 4096×2160 pixels, corresponding to common size used in cinematography. The images were preprocessed using 3-level reversible DWT transformation prior to their processing in EBCOT. Both benchmarks were run 30 times for the same configuration and codec.

Hardware and software configuration was as follows: CPU Intel Core i7 950 at 3.07 GHz, 6 GB DDR3 main memory, ASUS P6T6 WS Revolution motherboard, GeForce GTX 285 GPU (with 30 multiprocessors, 240 cores, 16 MB of shared memory, 2 GB of global memory). Software stack included Ubuntu Linux 9.04 with 2.6.28-15-server kernel, NVIDIA device drivers version 256.53, CUDA toolkit 3.1, and GCC version 4.3.3.

5.2 Experimental Results and Discussion

Table 2 summarizes results for both benchmark sets. It can be seen that for trivial small image (single color 512×512 image), the CPU implementations outperform GPU ones—this is caused by the overhead of memory transfers and low utilization of the GPU multi-processors. For non-trivial images and namely for larger images, the computation time prevails and the GPU implementations perform better compared to CPU ones. For efficient bpcuda implementation, even processing of 512×512 non-trivial images is approximately 2× better

[6] Because of shared memory size limitations, older NVidia GPUs are limited to 16×16 code blocks, while new NVidia Fermi architecture allows for larger code blocks.
[7] `http://www.openjpeg.org/`
[8] `http://www.ece.uvic.ca/~mdadams/jasper/`
[9] `http://www.kakadusoftware.com/`
[10] `http://cuj2k.sourceforge.net/`

compared to the best CPU implementation. Overall, 1.4–6.1 speedup can be observed for non-trivial images.

	OpenJPEG	**JasPer**	**Kakadu**	**CUJ2K**	**bpcuda**
Single-Color	39.9 ± 2.9	11.5 ± 2.3	1.2 ± 0.4	14.1 ± 0.1	12.4 ± 0.1
Lenna	128.9 ± 29.1	80.6 ± 20.2	47.8 ± 3.9	101.0 ± 0.2	26.3 ± 0.1
White-Noise	185.4 ± 4.9	129.9 ± 3.3	61.8 ± 3.9	98.2 ± 0.2	30.2 ± 0.1
1280×720	364.8 ± 2.9	164.0 ± 0.1	145.6 ± 4.9	120.1 ± 0.3	63.5 ± 0.3
1920×1080	723.3 ± 1.7	369.3 ± 16.6	309.7 ± 4.6	258.6 ± 0.4	137.2 ± 0.5
4096×2160	2818.0 ± 7.8	1481.5 ± 1.3	1093.1 ± 4.6	914.1 ± 0.8	662.9 ± 0.3

Table 2 EBCOT Tier-1 processing time [ms] of different implementations. Lower time means better performance.

To provide deeper insight into the EBCOT Tier-1 components, the profiling results of EBCOT Tier-1 of bpcuda and the reference CPU implementation JasPer are compared. We used the Valgrind suite for the application profiling JasPer and the combination of built-in CUDA timer functions for bpcuda. As shown in Fig. 2, the EBCOT Tier-1 is the most time-consuming part of the encoding chain on CPU. From the profiling information and the measured times, we can compare the runtimes of the single-threaded JasPer implementation and our bpcuda. In the case of JasPer processing the HD image (1920×1080 pixels), the context modeller occupies the 76 % (280.7 ms) and the arithmetic coder consumes 24 % (88.6 ms) of the EBCOT Tier-1. When bpcuda processes the same image, the context modeller consumes only 17 % (23.3 ms) and 83 % (113.9 ms) is spent in the arithmetic coder. The overall speedup 1.4–5.3 of the EBCOT Tier-1 is degraded due to yet not-optimized arithmetic coder. The speedup of the context modeller itself is 12 times when compared to JasPer, the best open-source CPU implementation. We consider the results of parallelized context modeller a significant improvement, indicating that we succeeded in reducing the EBCOT Tier-1 time-consumption mainly by re-formulation of the BPC part.

6 Conclusion and Future Work

In this paper, we have presented a novel approach to reformulating the context modeller algorithm of the EBCOT Tier-1 process in JPEG2000 in a way that enables an efficient implementation on GPU computing platform. The proposed algorithm has been implemented using CUDA, showing significant performance increase over existing CPU and GPU JPEG2000 implementations. In the future, we will focus on acceleration of the MQ-Coder in the EBCOT Tier-1 process and bit-stream formatting, thus finishing complete JPEG2000 acceleration for GPU architectures.

References

1 Tinku Acharya and Ping-Sing Tsai. *JPEG2000 Standard for Image Compression: Concepts, algorithms and VLSI architectures*. Wiley-Interscience, New York, 2004.

2 M. D. Adams. The JPEG-2000 still image compression standard. *ISO/IEC JTC 1/SC 29/WG 1*, 2412, 2001.

3 J. S. Chiang, Y. S. Lin, C. Y. Hsieh, et al. Efficient Pass-Parallel Architecture for EBCOT in JPEG2000. In *IEEE International Symposium on Circuits and Systems*, 2002.

4 C. Christopoulos, A. Skodras, and T. Ebrahimi. The JPEG2000 still image coding system: an overview. 46(4):1103–1127, Nov 2000.

5 A. Cohen, I. Daubechies, and J.-C. Feauveau. Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*, 45:485–560, 1992.

6 Ingrid Daubechies and Wim Sweldens. Factoring wavelet transforms into lifting steps. *J. Fourier Anal. Appl*, 4:247–269, 1998.

7 Joaquín Franco, Gregorio Bernabé, Juan Fernández, and Manuel E. Acacio. A parallel implementation of the 2D wavelet transform using CUDA. In *Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, pages 111–118, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

8 Rafael Gonzalez and Richard Woods. *Digital Image Processing*. Pearson/Prentice Hall, Upper Saddle River, 2008.

9 ISO/IEC 14492-1. Lossy/lossless coding of bi-level images, 2000.

10 ISO/IEC 15444-1. JPEG2000 image coding system—part 1: Core coding system.

11 C. J. Lian, K. F. Chen, H. H. Chen, and L. G. Chen. Analysis and Architecture Design of Block-Coding Engine for EBCOT in JPEG 2000. 13(3):219–230, mar. 2003.

12 M. W. Marcellin and A. Bilgin. JPEG2000 for digital cinema. *SMPTE Motion Imaging Journal*, 114(5-6):202–209, 2005.

13 M. W. Marcellin, M. A. Lepley, A. Bilgin, T. J. Flohr, T. T. Chinen, and J. H. Kasner. An overview of quantization in JPEG 2000. *Signal Processing: Image Communication*, 17(1):73–84, 2002.

14 J. Matela. GPU-Based DWT Acceleration for JPEG2000. In *Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, pages 136–143, 2009.

15 NVIDIA. *NVIDIA CUDA Programming Guide 3.0*. NVIDIA, 2010.

16 J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

17 In Kyu Park, Nitin Singhal, Man Hee Lee, Sungdae Cho, and Chris Kim. Design and Performance Evaluation of Image Processing Algorithms on GPUs. 99, 2010.

18 David S. Taubman and Michael W. Marcellin. *JPEG2000: Image Compression Fundamentals, Standards, and Practice*. Springer, 2002.

19 Armin Weiß, Martin Heide, Simon Papandreou, Norbert Fürst, and Ana Balevic. CUJ2K: a JPEG2000 encoder in CUDA, 2009.

20 Y. Z. Zhang, C. Xu, W. T. Wang, and L. B. Chen. Performance Analysis and Architecture Design for Parallel EBCOT Encoder of JPEG2000. 17(10):1336–1347, 2007.

Hijacking the Linux Kernel

Boris Procházka[1], Tomáš Vojnar[2], and Martin Drahanský[3]

1 **Faculty of Information Technology, Brno University of Technology**
Božetěchova 2, 61266 Brno, Czech Republic
`iprochaz@fit.vutbr.cz`

2 **Faculty of Information Technology, Brno University of Technology**
Božetěchova 2, 61266 Brno, Czech Republic
`vojnar@fit.vutbr.cz`

3 **Faculty of Information Technology, Brno University of Technology**
Božetěchova 2, 61266 Brno, Czech Republic
`drahan@fit.vutbr.cz`

Abstract

In this paper, a new method of hijacking the Linux kernel is presented. It is based on analysing the Linux system call handler, where a proper set of instructions is subsequently replaced by a jump to a different function. The ability to change the execution flow in the middle of an existing function represents a unique approach in Linux kernel hacking. The attack is applicable to all kernels from the 2.6 series on the Intel architecture. Due to this, rootkits based on this kind of technique represent a high risk for Linux administrators.

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.85

1 Introduction

We propose a new attack on the Linux kernel based on changing the control flow in the system call handler. The attack is applicable to all members of the 2.6 family on the Intel architecture. The main idea, changing the control flow in the middle of the system call handler, has not been to the best of our knowledge considered before and hence rootkits (tools setting up an environment for an attacker and hiding his/her activities) are not detectable using current detection tools. To compensate for the newly proposed attack, we also provide a new detection tool capable of detecting the new attack.

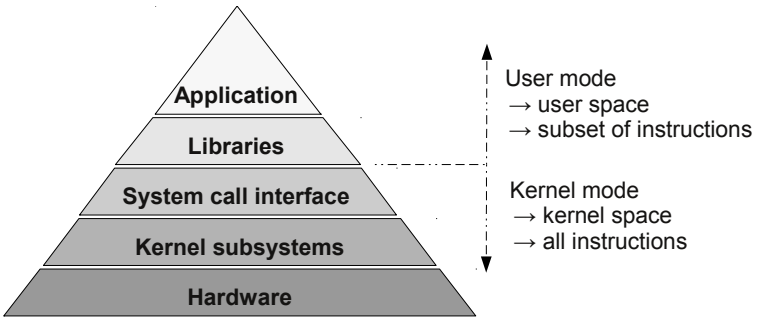


Figure 1 Operating system hierarchy

Basically, attacks can be divided into two main groups according to the operating system hierarchy (Fig. 1)[1, 2]:

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 85–92
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1. **Attacks on user mode.** Attacks against user's and system's applications and libraries. In this case, the attack is usually performed by a simple substitution of binaries, where the attacker swaps the originals with the corrupted. Fortunately, these kind of attacks are quite easy to detect thanks to checksums. There is also a possibility to use private, static-compiled binaries.
2. **Attacks on kernel mode.** Most of nowadays attacks are oriented towards kernel space, especially against kernel interfaces like system call interface or virtual file system. The main reason why these kind of attacks are so popular is because the attacker is able to gain control of the whole system with no mercy. We can easily imagine that if we change the behaviour of kernel interfaces, we will change the behaviour of the whole system (because user space programs rely on them). The attacker usually wants to hide his activity in the system so he modifies the interfaces to publish only a subset of real results. In the case of kernel space attacks, there is no reliable method how to reveal the attacker in the system. We can only hope that he was not skilled enough to masquerade all side effects of his activities.

In the rest of the paper, we will focus solely on the system call interface. We will discuss existing types of attacks on this mechanism and later on, an original attack will be revealed. Our idea will be to inject jump code in the middle of the system call handler.

2 System Call Interface

The system call interface forms an interface for switching between user and kernel mode. An application raises a query through the system call interface and the kernel tries to satisfy it. The system call interface is probably the most important interface in the system as it creates an abstract layer between users and the kernel.

In Linux (on the IA-32[1]), system calls are identified by numbers and their invocation is realized by a software interrupt. Parameters are passed through CPU's registers in a strict order: `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi` and `ebp`. The `eax` register is used to hold the system call number. Let us see how system calls work on the case of the `setuid()`[2] call (Fig. 2).

First of all, an application (or a wrapped routine in a library) has to fill CPU's registers with expected values. Then, an exception is risen by the `int $0x80` instruction which will cause the switch-over to the kernel mode and a system call handler activation. The address of the system call handler is saved in the interrupt table and is determined by an index into this table (`0x80` in this case).

The system call handler (implemented in `system_call()`) first saves the number of the system call and then the contents of all CPU's registers (`SAVE_ALL` macro). All passed arguments are saved and the CPU is released for further computing. Then, some basic tests are performed (we will cover this in detail in Subsection 4.1). If everything is alright, the system call number saved in the `eax` register is used as an index into the system call table to invoke a system call function (`sys_setuid()` in the described example).

System call functions perform the executive code. Their purpose is to change the system state or return system values. All system call functions are implemented as `asmlinkage`, which means that their arguments are saved exclusively on the stack.

When the system call function finishes, the return value is saved on the stack in the place of the `$eax` register. The system call handler continues with disabling interrupts

[1] Intel architecture, 32bit. Known also as i386 or x86.
[2] `Setuid()` sets the effective user ID of the current process.

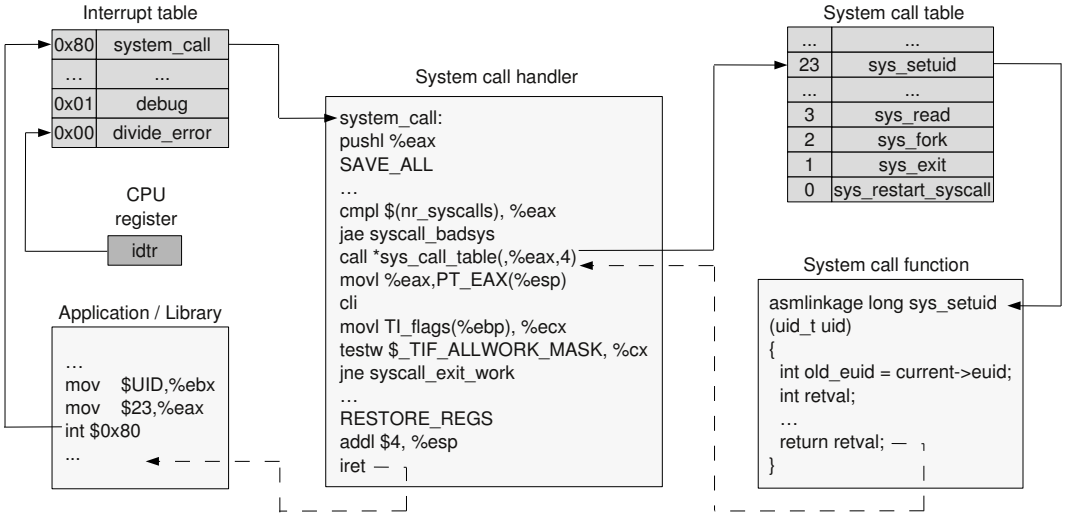


Figure 2 The Linux system call interface

and performing additional tests. On success, all CPU's registers are restored from the stack (`RESTORE_REGS` macro) and the function is finished with the `iret` instruction. This instruction causes a controlled switch-over back to the user mode and a continuation of the application.

The system calls are used very frequently and the implementation by software interrupt is not very efficient. Due to this, processors Intel Pentium II and older contain an additional instruction called a "fast system call" (the `sysenter` instruction). Although this instruction calls another handler (`sysenter_entry()`), the results (and even the body of the function) is almost the same as in the case of a system call handler (`system_call()`). So we do not have to distinguish between these two methods in the rest of the paper—the impact will be the same.

3 State of the Art

Attacks against the system call interface are relatively old and widespread. In this section, we will briefly describe existing types of such attacks:

1. **Attacks on the system call table.** The oldest and the most widely used way of intrusion. Its aim is to change an original record in the system call table with another version of the system call function [3]. This function is then used instead of the original. In most cases it acts like a wrapper for the original function filtering its results. The system call table is usually checked by administrators nowadays, so attackers had to develop more sophisticated ways of attacks.
2. **Attacks on the system call function.** If we do not want to change records in the system call table, we can move one step forward and change the prologue of the system call function [4]. The basic idea is to rewrite the entry point of the original function with a jump to a different function. The usage is the same as in the previous with one exceptions—if we want to use the original function, we have to repair its prologue or an infinite loop threatens.

3. **Attacks on the system call handler.** Another method how to redirect the execution flow without touching the system call table is to leave off using it. To do so, we have to copy the original system call table to a new location and change the pointer in the system call handler [5]. When it is done, the old system call table is not used anymore and we can modify our private one, just like in the first case.
4. **Attacks on the interrupt table.** If we take a closer look at Fig. 2, we can reveal that a second table is used in the subsystem—the interrupt table. The attacker can change records even in this table and forge the handler routine [6]. This attack is not trivial as the attacker needs to build up his own handler function and the interrupt subsystem is closely associated with the computer architecture.
5. **Attacks on the idtr register.** The interrupt table is located thanks to the `idtr` register. The value of the register can be modified by the `sidt` instruction. The attacker can do the same trick as in the attack on the system call handler—make a copy of the table and change the value in the `idtr` register to pointer on it [7].

4 The New Approach

In this section, we will focus on changing an execution flow in the middle of an existing function. The idea is motivated by attacks on the system call function, where the prologue is rewritten by a jump code. We will try to generalize this technique to be applicable even in the middle of functions.

If we want to hijack an execution flow in the middle of an existing function, we have to rewrite its code. This is quite easy. The biggest problem is to ensure the original behaviour of the corrupted function. If we write down all the problems we have, we will get these three issues:

1. **Seven bytes of space.** For hijacking the execution flow, we have to rewrite the existing code with a jump or call instruction. The easiest way is to fill one of the CPU's registers with the destination address and then perform an absolute jump[3]. If we write it down in assembly, we will get something like this:

   ```
   movl $0,%eax --> \xb8\x00\x00\x00\x00
   jmp  *%eax   --> \xff\xe0
   ```

   The code is compiled as shown on the right side. The result is seven bytes long machine code. This means that we will need to rewrite at least two instructions as the Intel architecture uses a variable instruction length.
2. **Keep valid code.** We have to keep the code valid after overwriting it. If we produce an invalid instruction, the CPU rises an exception and immediately terminates the process. Due to this, we have to respect the beginnings and ends of instructions and do not overwrite code containing the labels.
3. **Keep the original semantics.** We will change the structure of the considered function by injecting some code. As we want to keep the original behaviour, we have to compensate the rewritten code to sustain the original semantics. This is the most difficult condition and requires a data analysis because when the hijacking is completed, the function must continue in its execution with no restrictions.

[3] It is not possible to do an absolute jump by `jmp $address`.

We will now demonstrate how to solve the above problems for the particular case of the system call handler.

4.1 Where to Hijack Control Flow in the System Call Handler

In this subsection, we will study the code of the system call handler and try to determine best places for hijacking. The handler is a low-level subsystem thus it is completely written in assembler[4]:

```
system_call:
  pushl %eax                        //Storing of system call number
  SAVE_ALL                         //Storing of all CPU's registers
  movl $0xffffe000, %ebx            //Calculation of the pointer to
  andl %esp, %ebx                                 //current process
```

The function starts its activity by storing the system call number and all CPU's registers in the stack. Afterwards, a pointer to the current process is calculated and saved in the `ebx` register.

In a above code fragment, we now try to find a candidate place for hijacking. We cannot rewrite the beginning instructions which are saving data from the user space (we would probably loose some data). However, when all data from the user space is saved, the CPU is released and there is an ideal opportunity for hijacking the control flow. If we measure the number of bytes of two instructions calculating the pointer to the current process (`movl`, `andl`), we will get seven bytes. The calculation of the pointer is also standalone and independent and it can be easily reproduced.

```
  testw $_TIF_WORK_SYSCALL_ENTRY,TI_flags(%ebp)  //Process traced?
  jnz syscall_trace_entry           //If so, jump to trace function
  cmpl $(nr_syscalls), %eax       //eax >= number of system calls?
  jae syscall_badsys                                  //If so, abort
```

The system call handler continues with two tests. The first one checks whether the running process is being traced. If the trace flag is set, the process is stopped and made available to the debugger. The second test checks the validty of system call number in `eax` register. As the number in `eax` register represents an index into the table, it cannot be greater than total number of system calls in the system.

We can leave studying of the fragment above very briefly. We would break code containing relative jumps (`jnz`, `jae`) which would be very difficult to compensate.

```
  call *sys_call_table(0, %eax, 4)   //Calling sys_call_table[eax]
  movl %eax,PT_EAX(%esp)                 //Storing of return value
```

The core of the system call handler. The `eax` register is used as an index into the system call table to call the system function. The return value is saved into the stack in the position of the `eax` register for the user space.

Despite the core of the system call handler is suitable for hijacking, we will leave it off. The reason is that the pointer to the system call table is also modified by attacks on the system call handler and the system scanners generally test this value.

[4] This code can be found in Linux kernel source: `arch/x86/kernel/entry_32.S`.

```
cli                                          //Clear Interrupts
movl TI_flags(%ebp), %ecx              //Copy process flags in ecx
testw $_TIF_ALLWORK_MASK, %cx              //Is needed extra work?
jne syscall_exit_work                       //If so, do extra work
```

When the system function returns, all interrupts are masked and the process is tested against additional work requirements (unserved signal, process is traced).

The code fragment above offers another opportunity for hijacking. The length of the `movl` and `testw` instructions is eight bytes, which is enough for jumping out. These two instructions are also standalone so we can reproduce them.

```
RESTORE_REGS                            //CPU's registers restoration
addl $4, %esp          //Clearing up system call number from stack
iret                                          //Return from interrupt
```

The end of the system call handler prepares the system to switch back to the user mode. All CPU's registers are restored from the stack (filled with results of the system call), the system call number is cleared and the `iret` instruction is triggered.

The hijacking of this terminating fragment is possible too. The main problem is when the process is traced and so does not use this code fragment.

4.2 Changing the Control Flow in the System Call Handler

So far, we have located two suitable places for hijacking the control flow in the system call handler. Both of them are occurring in the whole `2.6` kernel and thus offering a very good portability and usability.

Our goal is to modify the return values from the system call functions which are available after their invocations. Due to this, we will focus on the second place suitable for hijacking the control that we have identified in the system call function.

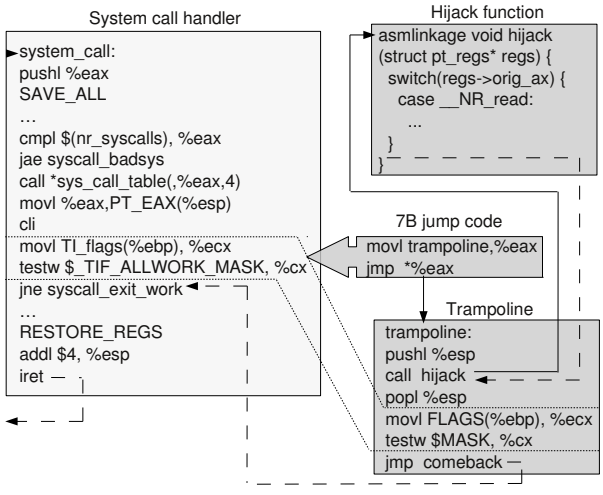


Figure 3 Hijack implementation

We will rewrite selected part of the code by an unconditional jump into the trampoline function (`trampoline()`). The trampoline has several tasks. First, it saves the top of the stack to ensure a convenient access to the system call results. Subsequently, it calls the hijack function (`hijack()`), which modifies these results on the basis of the system call number

(deleting records about hidden directories in the `ls` program, for example). When the hijack function returns the execution back to the trampoline, the system has to be set back to the original state. To do so, the stack is cleared, instructions used for the hijacked jump are compensated and the execution is returned just after the kidnapped place. The attack is over.

5 Experiments

To verify all the presented facts and ideas, two rootkits based on this new technique were implemented. The first one, called *MoleKit*, is able to infiltrate the system through writing into the `/dev/mem` file[5]. Molekit provides only basic services like process and directory hiding. It is because attacking the system through `/dev/mem` file is quite complicated due to many heuristics needed (finding code patterns in the memory), but it represents a way how to infiltrate a system even without a loadable kernel module support (see [8] for more information).

The second rootkit is called *Powerkit* and infiltrates the system using a kernel module. The main advantage of kernel modules is the access to the kernel API. This makes it possible to implement advanced features such as keylogging or escalation of privileges.

As these two rookits use the new method of hijacking, no current anti-rootkit or validity scanner can detect them. Because of that, we implemented *Sentinel* scanner [9]. Sentinel is a tool which periodically checks integrity of the interrupt subsystem, the system call interface (including the new method presented in this paper) and the virtual file system. The detection is based on testing key values of these subsystems (tables, pointers to tables, system call handler code, function prologues, ...) against their reference values obtained after system instalation.

6 Conclusion

In this paper, a new method of hijacking the Linux kernel was presented. The attack was successfully verified on the whole 2.6 kernel series and two rootkits based on this new technique were implemented. Because these two rookits would represent a serious security risk for Linux administrators, a tool for their detection was published.

Acknowledgements This work was supported in part by the the Czech Ministry of Education (projects MSM 0021630528) and the internal BUT FIT grant FIT-10-1.

References

1 R. Love. *Linux Kernel Development*. Novell Press, Indiana 46240, USA, 2006, ISBN 0-672-32720-1

2 P.D. Bovet, M. Cesati. *Understanding the Linux Kernel*. O'Reilly, USA, 2005, ISBN 0-596-00565-2

3 P. Sobolewski. *Hakin9 Nr 2/2005*. Software-Wydawnictwo Sp. z o.o., Warszawa, Poland, 2005, ISBN 1214-7710

4 S. Cesare. *SYSCALL REDIRECTION WITHOUT MODIFYING THE SYSCALL TABLE*. http://www.ouah.org/stealth-syscall.txt

[5] It is a character device providing access to the main memory

5 Devik, Sd. *Linux on-the-fly kernel patching without LKM*. http://www.phrack.org/issues.html?issue=58&id=7#article

6 Kad. *Handling Interrupt Descriptor Table for fun and profit*. http://www.phrack.org/issues.html?issue=59&id=4#article

7 B. Prochazka. *Methods of Linux Kernel Hacking*. FIT BUT, Brno, 2008, bachelor's thesis

8 A. Lineberry. *Malicious Code Injection via /dev/mem*. http://www.blackhat.com/presentations/bh-europe-09/Lineberry/BlackHat-Europe-2009-Lineberry-code-injection-via-dev-mem.pdf

9 B. Prochazka. *Program Sentinel*. http://www.stud.fit.vutbr.cz/ xproch63/conference/memics2010/sentinel.zip

Fast Translated Simulation of ASIPs

Zdeněk Přikryl, Jakub Křoustek, Tomáš Hruška, and Dušan Kolář

Brno University of Technology, Faculty of Information Technology
Božetěchova 2, 612 66 Brno, Czech Republic
{iprikryl, ikroustek, hruska, kolar}@fit.vutbr.cz

Abstract

Application-specific instruction set processors are the core of nowadays embedded systems. Therefore, the designers need to have powerful tools for the processor design. The tools should be generated automatically based on a processor description. One of the most important tools is the simulator. It is used during a testing phase of the processor design and during target software development. The key feature of the simulator is its speed. The concept of a special simulation type – translated simulation – is presented in this paper. This simulation exploits information from a target C compiler. Both the simulator and the C compiler are generated based on the processor description in an architecture description language ISAC. Experimental results of this concept show very good simulation speed and fast generation of the simulator.

Keywords and phrases Hardware/sofware co-design, simulation, architecture description languages, application-specific instruction set processors.

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.93

1 Introduction

Embedded systems have become essential part of our nowadays lives. One can find them almost everywhere. There can be one or more application-specific instruction set processors (ASIPs) inside an embedded system. Each processor has usually dedicated functionality and it is highly optimized for it. There are many trade-offs among which part of functionality should be implemented directly in the processor and which part should be implemented in software. The process of optimal solution searching is called design space exploration (DSE). Therefore, the designer should have a good integrated desktop environment (IDE) for the processor design. The IDE should provide automatic tool-chain generation based on the processor description. The tool-chain consists of the tools for processor programming, such as an assembler or C compiler, and of the tools for processor simulation, such as a simulator or profiler.

The processor itself can be described using either hardware description language (HDL) or architecture description language (ADL) (see [11]). Generally, ADLs are better for fast DSE and rapid processor prototyping, since ADL hides hardware details. Those details can be unknown at the beginning of the processor design or the designer does not want to take care of them.

One of the tools used during the whole processor design is a simulator. Therefore, the simulator has to be fast enough. Furthermore, the simulator is also used for the target software development (often at the same time as the hardware is designed – hardware/software co-design). There are several different types of simulators. Each of them is usually used in different phase of the processor design (less accurate simulator during the first steps in DSE, more accurate simulator during the preparation of final hardware realization). Various types are discussed in the section 2, and advantages or disadvantages are highlighted.

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 93–100

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Our project running at Brno University of Technology is called Lissom (see [7]). It provides whole IDE for the processor design and tools for multiprocessor system on a chip design. In this paper, we present the concept of a block-accurate simulator called *translated* simulator. It uses additional information of basic blocks from the target C compiler. It is based on the LLVM (Low Level Virtual Machine) platform [8]. The compiler can be also generated based on the processor description using the ISAC (Instruction Set Architecture C) language. The ISAC language is inspired by the LISA (Language for Instruction Set Architectures) language [5] and it has been developed within the Lissom project.

2 State of the Art

There are a few projects which try to give the developer a whole IDE for the processor design. Each of them uses its own description language which has been developed within the project. An open source project ArchC [1] uses ADL called ArchC. It is a description language for pipeline systems based on SystemC. The processor description is composed of several parts. The designer can describe resources, such as memories or registers, instruction set and its behavior. The behavior is described with SystemC functions in shared libraries.

Another widely used ADL is LISA. The processor description in LISA language is composed of several parts. In one part, resources are defined. In the other part, an instruction set with behavior and processor microarchitecture is described. In both projects (ArchC and LISA), the interpreted and compiled simulators are available, but none of them supports translated simulator.

At the Vienna University of Technology, an ADL called xADL (see [2]) was developed. The processor is described with hardware blocks which are interconnected. The xADL language supports generation of the translated simulator. The LLVM platform is used for simulator creation; therefore, the creation of simulator takes a long time since the whole LLVM has to be compiled.

An introduction to the simulator terminology used in this paper is in the following text. The basic type of simulator is an interpreted simulator. The run of interpreted simulator is based on the following concept. It fetches an instruction then it decodes the instruction, and executes it. Therefore, it is not dependent on simulated application and it allows self-modifying code out of the box. On the other hand, it constantly fetches and decodes the same instructions (e.g. instructions within a loop). Hence, this slows the simulator down.

Another type is a compiled simulator. Unlike the interpreted simulator, the compiled simulator is created in two steps. In the first step, a simulated application is analyzed. In the second step, based on the analysis, the simulator itself is created. It is clear that the basic type of compiled simulator cannot simulate self-modifying code and it is dependent on the analyzed application. Note that this is not true in the case of dynamic compiled simulators (see [12]).

Other important feature of simulators is the simulator accuracy. Basically, the simulator can be cycle-accurate, instruction-accurate or block-accurate. In the first type, the basic step of the simulation is single clock cycle. Therefore, this type of simulator is very close to hardware and gives the most relevant information about the behavior of a real processor. On the other hand, the speed is not very good, since the whole microarchitecure is simulated. Therefore, this type of simulator is used when the processor design is stable enough.

In the second type, the basic step of the simulation is single instruction. The processor microarchitecture is not simulated. This type is used for target software development (i.e. the software which will run on designed processor), or it can be used for virus detection where

an instruction-accuracy is enough. Note that the cycle-accurate and instruction-accurate simulators can be either interpreted or compiled [12].

The block-accurate simulator uses a whole basic block in a simulated application as a basic step of the simulation. The basic block is an indivisible sequence of instructions with one entry point (start address), one exit point (end address), and no branch instructions within it. These addresses cannot be always determined during static analysis of a simulated program. There can be (and in a real processors usually is) a branch instruction, which gets a destination address from a register. Therefore, this address is known only during a simulation. The start and end addresses of all basic blocks are known during a compilation, so the compiler can save this information for further usage by simulator. Since we need to preprocess this information, the block-accurate simulators are only compiled; therefore they are dependent on particular simulated application.

3 ISAC Language

The ISAC language falls into so-called mixed architecture description languages. It means that the processor instruction-set with processor microarchitecture is described in one model. The processor model consists of two parts in the ISAC language. In the first part, the processor resources, such as registers or caches, are described. In the second part, processor instruction-set together with microarchitecture is described. The basic construction of the second part is *operation* construction. The operation can have several sections. The section describes either instruction-set or microarchitecture and forms one of the four basic models of processor. Each model describes the processor from different point of view. The models are: *instruction-set* model, *timing* model, model of *instruction analyzers hierarchy* and *behavioral* model.

The instruction-set model is formed by the *assembler* and *coding* sections. The assembler section describes the textual form of an instruction (assembly language). The coding section describes the binary form of the instruction (machine code). The timing model is formed by the *activation* section. This section denotes what and when is done in the microarchitecture of processor (e.g. timing of processor pipeline). The model of instruction analyzers hierarchy is formed by the section *structure*. This section describes timing of instruction decoding. The behavior model is specified by sections *expression* and *behavior*, where the ANSI C language is used (i.e. ANSI C descibes the behavior of instructions and processor microarchitecture). Note that the expression section has the same meaning as the return statement in a function (i.e. it is used for returning of a value if particular operation is used during instruction decoding).

Operations can be grouped according to some criteria, such as similar functionality (e.g. operations describing arithmetic instructions). The *group* construction is used for grouping of that operations or other groups. An operation can use other operations or groups using the *instance* statement (e.g. an operation describing move instruction uses another operation describing immediate operand). The processor model consists of a resource description and many operations and groups. There is one mandatory operation called `main`. This special operation is used for synchronization (i.e. clock cycle generation). An example of two operations is in Listing 1. There is an operation describing 8-bit immediate operand using 8-bits attribute and other operation describing `move_acc` instruction. The second operation uses results from previous operations (i.e. it uses results from expression sections which is the value of immediate operand). More information can be found in [9].

Basically, the processor can be described on instruction-accurate or cycle-accurate level

Listing 1 Example of ISAC Language Source Code

```
// Operation with one attribute
// attr for 8 bit operand
OPERATION imm8 {
  ASSEMBLER { attr=#U };
  CODING { attr=0bx[8] };
  EXPRESSION { attr; };
}
// Operation describing move
OPERATION move_acc {
  INSTANCE imm8;
  ASSEMBLER { "move_acc" imm8 };
  CODING { 0b0101 imm8 };
  BEHAVIOR { acc = imm16; };
}
```

by the ISAC language. Note that the processor model at instruction-accurate level has operation `main` with the structure section.

4 Concept of Translated Simulation

The following notation is used in this section. A target C compiler is the generated C compiler. It is generated from the processor model and it is based on the LLVM platform. A target application is the application which will run on the designed processor. A host C compiler is *gcc* compiler which compiles the generated simulator itself. The process of the simulator generation has three parts. The processor description has to be on the instruction-accurate level. The first part is performed only once for any particular processor description. The next two parts are target application specific, so they have to be performed every time when the target application is changed.

In the first part, the analyzer of the target application is generated. It is generated only once and it is based on the processor description (i.e. it does not have to be re-generated until the processor description has changed). This analyzer is similar to the disassembler, so it accepts an application in the machine code. But instead of emitting the assembly code, it emits C code. The analyzer itself is based on the enhanced formal models *coupled finite automata* [6] and *lazy finite automata*. *Lazy finite automaton* M is septuple $M = (Q, \Sigma, \delta, s, F, S, z)$, where Q is a finite set of states, $\Sigma = \{0, 1\}$ is an input alphabet, $s \in Q$ is a starting state, $F \subseteq Q$ is a set of final states, $\delta = Q \times \Sigma^* \times Q$ is a finite transition relation, S is a set of *semantic actions*, and z is a relation $z \subseteq \delta \times S$. The relation z assigns semantic actions to transition relations. The semantic action is indivisible sequence of a C code which is executed when a particular transition is taken. Definitions of configuration, move, and accepted language are analogical to definitions in normal lazy finite automaton.

Coupled finite automata C used in the analyzer is a triple $C = (M_1, M_2, h)$, where M_i is a lazy finite automaton for $i = 1, 2$, and h is a bijective mapping from δ_1 to δ_2. Definition of bijective mapping h, and translation by coupled finite automata are analogical to definitions in normal coupled finite automata (see [6] for more details). The automaton M_1 is used as an instruction parser ($\Sigma_1 = \{0, 1\}$) and the automaton M_2 is used as a C code generator. The set S_2 contains the modified C code from the behavior and expression sections, which are taken from the processor description. The content of the behavior section is changed in a way that the constants, which are represented by attributes, are replaced by their evaluation (values

are obtained from the automaton M_1 during translation). Furthermore, each statement is encapsulated, so the C code is only printed into a file, not executed. Let's assume the behavior section from the Listing 1. In the simple case, the content of original expression section of the *imm8* operation, `attr;`, is changed to `fprintf(fp, "imm8 = %d\n", attr);`, and the content of original behavior section of the *move_acc* operation, `acc = imm8;`, is changed to `fprintf(fp, "acc = imm8;");`.

In the second part, the core of translated simulator is created (i.e. the analyzer generates a C code based on the target application). The generated output C code has to be organized somehow. If the output would be only one single function, then, in the the case of large target application, the function would become uncompilable (e.g. problems with optimizations, problems with virtual memory, etc.). Therefore, the address space of the designed processor is divided into so-called segments. Each segment has the same fixed size, which is set during the creation of an analyzer by the developer. The size has to be equal to some power of two (e.g. 512 or 1024). The reason for that action is explained later. Functions are generated for each segment. It simulates instructions within the segment. This function has one parameter. It is used for passing the program counter. Each function contains single `switch` statement which takes this parameter. The `case` statement is generated for each instruction within the segment. Note that the `case` bodies are generated by the M_2 automaton. In a straight approach, each `case` is ended with a `break` statement. There are two main reasons why this approach does not allow effective host compiler optimization. Firstly, each `break` ends the function. That means that the computed values, which can be used in the next simulated instruction, are swapped out from the host registers to the main memory. From the host processor point of view, it would be better to keep these values in the registers. Secondly, the `case` does not allow additional optimizations since it creates the end of basic block in the simulator code. The side effect of the two mentioned constructions leads to worse cache hit/miss ratio too. Therefore, the following improvement is used.

Since the analyzer knows the starting and ending addresses of basic blocks in the target application (they are stored as debug info in the target application), the `break` is generated only if an address of an analyzed instruction is equal to an ending address of some basic block. Otherwise, the simulation of new clock cycle is performed (i.e. the behavior section of `main` operation is executed). The `case` is generated only if an address of an analyzed instruction is equal to a starting address of some basic block. The Fig. 1 shows previous principle. Each instruction has 32 bits and the address space can be addressed by 8 bits in

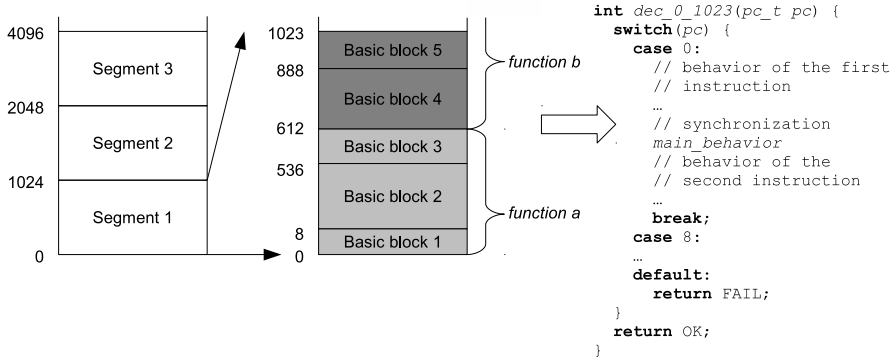


Figure 1 Principle of a translated simulator generation

this example. Note that there is no `break` for address 0 (it is not ending address) and there it no `case` for the address 4 (it is not starting address). Therefore, the unmodified behavior section of synchronize operation `main` is generated there.

The whole target application is represented by several functions. These functions are stored in a table. The key to this table is created by the right bit shift of an instruction address value. The count of bits needed for shifting is computed from the segment size (square root of segment size). The limitation of the segment size (power of two) guarantees a fast transformation from the addresses to the keys used in the table. Note that the valid addresses are addresses of the start and end of basic blocks. The simulator itself is formed by a loop which calls particular functions from the table together with the execution of the unmodified behavior section of synchronize operation `main`.

In the third part, the simulator itself is created via a compilation of target application independent parts, such as the representation of the resources, and target application dependent part (i.e. functions generated by the analyzer).

5 Experimental Results and Future Research

Several experiments of translated simulation concept were performed. As testing processor architectures we chose MIPS and VEX. Both processors are described on instruction-accurate level in the ISAC language. MIPS is a 32bit RISC (reduced instruction set computer) architecture developed by MIPS Computers Systems. The instruction-set of MIPS is in version MIPS32 Release 1. VEX is a four-slot 32bit VLIW (very large instruction word) architecture designed by HP [4]. Each slot is unique (i.e. each slot processes different types of instructions).

MiBench test suite [10] was used for testing and simulation speed measuring. All simulations were performed on the same host – Intel Core 2 Quad with 2.8 GHz, 1333 MHz FSB and 4GB RAM running 64-bit Linux based operating system. The gcc (v4.4.4) compiler with optimizations (–O3) was used for creation of simulator generators and simulators. All results are the average values of several runs of each test (differences of values from average are in tenths of a percent).

Fig. 2 shows the performance comparison of all simulator types for MIPS and VEX. As we can see in this figure, the speed of translated simulation is approximately 70% faster than compiled simulation and up to four times faster than interpreted simulation.

In average, the times needed for creation of compiled simulator generators are 5.03s for MIPS and 17.42s for VEX. The creation of translated simulator generators takes 4.93s

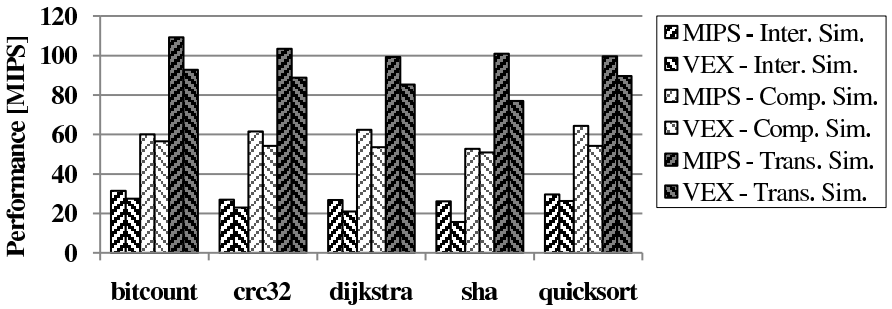


Figure 2 Performance comparison of all simulator types

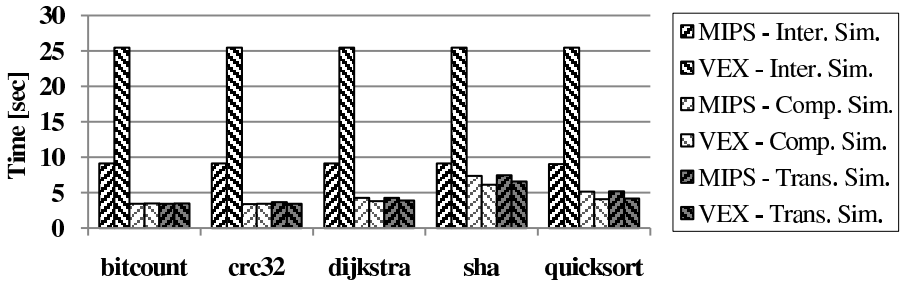


Figure 3 Simulator generation time

for MIPS and 17.17s for VEX. In Fig. 3, we can see generation times of simulators based on a target application (for the compiled and translated simulation). The time needed for generation of an interpreted simulator is constant because it is application independent. The sum of times needed for creation of translated simulator generator and simulator itself is lower than creation time of interpreted simulator, based on target application complexity. This is another advantage of this simulator type.

Our concept of translation simulation is fully competitive. For example, our solution is in average 40% faster than the concept of translated simulation created at Vienna University of Technology (according to results in [3]). The comparison was made on MIPS architecture and the set of five MiBench algorithms.

6 Conclusion

The concept of translated simulation is presented in this paper. The simulator is generated based on a processor description and a target application. A processor is described using the architecture description language ISAC. The generator of simulator needs to know all starting and ending addresses of all basic blocks in the target application. This information is obtained from the C compiler. It is based on LLVM platform and it is generated from the same processor description. The generator of a simulator is based on several formal models. The same formal models are also used in other generators, such as hardware description generator. Hence, no additional huge verification of hardware realization is needed. The experimental results show very good simulation speed and the time needed for a creation of the simulator itself is low. All mentioned features provide the powerful platform for ASIP and target software development.

Acknowledgements This work was supported by the research funding MPO ČR No. FR-TI1/038, BUT FIT grant FIT-S-10-2, doctoral grant GA ČR 102/09/H042, SMECY, and by the Research Plan No. MSM 0021630528.

References

1 ArchC Architecture Description Language. http://archc.sourceforge.net/

2 Brandner, F.: Compiler Backend Generation from Structural Processor Models. PhD thesis, Institut für Computersprachen Technische Universität Wien (2009)

3 Brandner, F.: Fast and Accurate Simulation Using the LLVM Compiler Framework. In RAPIDO '09: 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (2009)

4 Fisher, J. A., Faraboschi, P., Young, C.: Embedded Computing – A VLIW Approach to Architecture, Compilers, and Tools. Morgan-Kaufmann Elsevier Publishers (2005) 978-1-55860-766-8

5 Hoffmann, A., Meyr, H., Leupers, R.: Architecture Exploration for Embedded Processors with LISA. Kluwer Academic Publishers (2002) 4020-7338-0

6 Hruška, T., Kolář, D., Lukáš, R., Zámečníková, E.: Two-Way Coupled Finite Automaton and Its Usage in Translators. In: New Aspects of Circuits, Heraklion, GR, WSEAS (2008) 445-449

7 Lissom Project. http://www.fit.vutbr.cz/research/groups/lissom/

8 LLVM Compiler Infrastructure. http://llvm.org/

9 Masařík, K.: System for Hardware-Software Co-Design. FIT BUT, Brno, CZ (2008)

10 MiBench test suite. http://www.eecs.umich.edu/mibench/

11 Mishra, P., Dutt, N.: Processor Description Languages. Morgan Kauffman Publishers (2008) 978-0-12-372487-2

12 Přikryl, Z., Hruška, T., Masařík, K., Husár, A.: Fast Cycle-Accurate Compiled Simulation. In PDeS '10: 10th IFAC Workshop on Programmable Devices and Embedded Systems (2010)

Test-Case Generation for Embedded Binary Code Using Abstract Interpretation

Thomas Reinbacher[1], Jörg Brauer[2], Martin Horauer[3], Andreas Steininger[1], and Stefan Kowalewski[2]

1 **Embedded Computing Systems Group, Institute of Computer Engineering, Vienna University of Technology, Treitlstrasse 3, A-1040 Vienna, Austria**
{treinbacher, steininger}@ecs.tuwien.ac.at

2 **Embedded Software Laboratory, RWTH Aachen University, Ahornstraße 55, D-52074 Aachen, Germany**
{lastname}@embedded.rwth-aachen.de

3 **Department of Embedded Systems, University of Applied Sciences Technikum Wien, Höchstädtplatz 5, A-1200 Vienna, Austria**
horauer@technikum-wien.at

Abstract

This paper describes a framework for test-case generation for microcontroller binary programs using abstract interpretation techniques. The key idea of our approach is to derive program invariants a priori, and then use backward analysis to obtain test vectors that are executed on the target microcontroller. Due to the structure of binary code, the abstract interpretation framework is based on propositional encodings of the program semantics and SAT solving.

1998 ACM Subject Classification C.3, D.2.4, D.2.5

Keywords and phrases Test-Case Generation, Embedded Binary Code, Abstract Interpretation

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.101

1 Introduction

Traditionally, formal verification and structural testing are considered as orthogonal concepts for increasing the quality of software. Whereas formal verification techniques such as model checking or abstract interpretation establish a full proof of correctness, testing increases confidence in the correctness of a system by meeting certain coverage criteria, where none of the examined paths violates the specification. However, the underlying coverage criteria, which are often dictated by industrial standards [20], are typically insufficient for finding property violations as argued by Heimdahl et al. [13].

In the embedded systems domain, verification and validation techniques should ideally be applied to the executable binary code of a program, since the exact semantics of the program is not unambiguously specified in high-level representations such as C code [1]. Further, it is not unknown for compilation itself to introduce errors [10]. However, embedded systems code often strongly relies on the behavior and state of the hardware and on interaction with the environment. The need to model these two properties properly, among others, aggravates the state explosion in model checking and limits its applicability. On the other hand, abstract interpretation provides a scalable approach to verification that often suffers from imprecision, and subsequently, a high number of spurious warnings. This is even more so on the binary-code level, where interleavings of arithmetic and logical operations as well as the finite precision of registers pose additional challenges.

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 101–108

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In case of a violated property, abstract interpretation typically does not provide a counterexample, which is extremely helpful for fixing the defect [7]. By way of contrast, this property is fulfilled by both model checking and testing.

1.1 Approach

The ultimate goal of our work is to derive real counterexample traces for binary programs. To do so, our approach uses abstract interpretation to detect potential violations, and then derive paths through the program that could have led to that violation using backward analysis. These paths define test vectors, which are examined on the real hardware to filter spurious traces that have been introduced through over-approximation.

1.2 Contributions

Spurious warnings are a major issue when applying abstract interpretation in industrial practice. Typically, investigating spurious warnings relies on manual inspection of program invariants. The complex structure of embedded code makes manual inspection difficult and time-intensive. To leverage these issues in embedded-software verification, we contribute a framework that: (i) applies abstract interpretation to generate assertion-directed test cases; (ii) provides a link to the actual target hardware; (iii) automatically identifies spurious test traces.

2 Test-Case Generation Using Abstract Interpretation

Our framework (cf. Fig. 1) takes an executable binary file and a specification (cf. Sect. 2.1) as inputs. The binary file is ready to be run on the target hardware. After parsing, we build an initial control flow graph (CFG) of the binary and apply abstract interpretation (cf. Sect. 2.2) to derive program invariants. These invariants are used by the test-case generator to identify possible specification violations. Then, a backward analysis derives actual program inputs (cf. Sect. 2.3), that drive execution towards the specification violation. The test traces are then transferred to and executed on real hardware (cf. Sect. 2.4), i.e., an IP-core instance of the target microcontroller running within an FPGA embedded in its operating environment. A test-case monitor is attached to the IP core that tracks specification items during execution and provides runtime feedback.

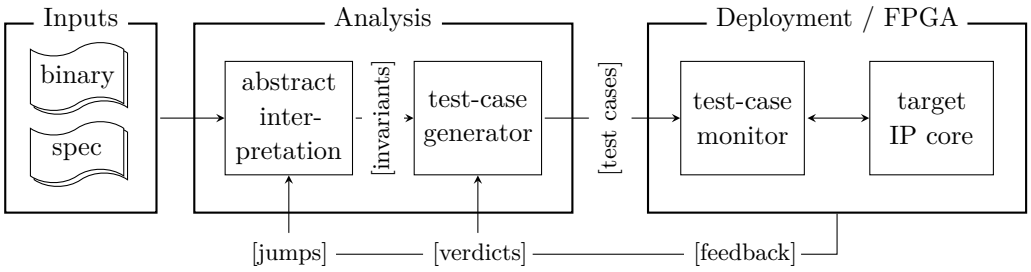


Figure 1 Framework overview

2.1 Specification Language

In the past, we have carried out a case study [18] in cooperation with an industry partner using [MC]SQUARE [21], which is a binary code verification tool. When confronting our partner with the full expressive power of temporal logics (CTL in this case), it turned out that it is particularly difficult for test engineers to translate their well-understood textual specifications into temporal logic formulas. Moreover, most specification items of the case study were local assertions (properties that hold at a specific program location) or global invariants (properties that hold at any program location), an observation also emphasized by Hoare [14, p. 10]. Consequently, to express program properties of interest, we propose a simple specification language, which is defined through the following grammar:

$$\begin{array}{lcl} \Psi & ::= & \mathcal{A}(pc, \varphi) \mid \mathcal{I}(\varphi) \\ \varphi & ::= & \mathsf{true} \mid \mathsf{false} \mid \neg\varphi \mid \varphi_1 \ \wedge \ \varphi_2 \mid \varphi_1 \ \vee \ \varphi_2 \mid \mathsf{AP} \end{array}$$

To express the semantics of this specification language, let a state of a program be a tuple $\langle pc, m\rangle \in \mathsf{Locs} \times \mathsf{Mem}$, where Locs is a finite set of program locations, and Mem represents the set of all possible memory configurations of the microcontroller. Then, the state space of the program is a subset of $\mathsf{Locs} \times \mathsf{Mem}$. The property φ is a predicate over memory locations $m \in \mathsf{Mem}$. Additionally, AP denotes the set of atomic propositions about memory cells in Mem. The satisfaction relation associated with φ is intuitively clear, following the standard inductive definition. If $m \in \mathsf{Mem}$ satisfies φ, we write $m \models \varphi$.

Properties, in turn, can be of local or global nature. A *local* assertion is a property $\mathcal{A}(pc, \varphi)$ attached to a certain program location $pc \in \mathsf{Locs}$. Given a set of states $S \subseteq \mathsf{Locs} \times \mathsf{Mem}$, then $\mathcal{A}(pc, \varphi)$ holds w.r.t. S iff $m \models \varphi$ for all $\langle pc', m\rangle \in S$ with $pc = pc'$. Similarly, a *global* invariant $\mathcal{I}(\varphi)$ holds iff $m \models \varphi$ regardless of pc'.

Our framework either reads a user-defined specification or uses existing assertions from the high-level representation of the program by parsing compiler-generated debug information.

2.2 Abstract Interpretation

The key idea in abstract interpretation is to simulate the execution of each concrete operation $g : C \to C$ in a program using an abstract analogue $f : D \to D$, where C and D denote the domains of concrete values and descriptions. Each abstract operation f is designed to model its concrete counterpart g in the following sense: If $d \in D$ describes a concrete value $c \in C$, then the result of applying g to c is described by applying f to d. Typically, the abstract operations are designed manually. However, handcrafting transformers for the complete instruction set of a microcontroller, which consists of more than 100 instructions, is time-consuming and error-prone. Consequently, we synthesize optimal transfer functions [19] from propositional encodings of the instructions' semantics using SAT solving [4]. The process of translating instructions into propositional Boolean formulas is often colloquially referred to as *bit-blasting*.

To derive a set of test cases, our abstract interpretation framework first computes invariants using intervals and synthesized transformers. If the invariants exhibit a potential property violation, we use backward analysis to derive a path (the test case) from the property violation to the start of the program. It is important to observe that sound abstract interpretation itself requires a CFG of the program to be available. However, recovering indirect control from binaries is a notoriously difficult problem [16]. Consequently, the CFG used in the abstract interpretation framework is incrementally extended using information gained through the test-case execution. Since the aim of our work is to detect test traces that

exhibit faulty behavior instead of proving correctness of an implementation, this approach is convenient. The remainder of this section discusses two approaches used to derive program invariants.

2.2.1 Affine transfer functions of basic blocks

The semantics of a microcontroller instruction can be encoded in propositional logic, which has become a standard technique in software verification, owing much to the advances in bounded model checking [3]. To illustrate this, consider the instruction `INC A` on an 8 bit architecture, which increments register `A` by one. The input and output values of `A` are represented by bit-vectors of length 8, denoted $\boldsymbol{a}$ and $\boldsymbol{a}'$, respectively. Then, the effects of applying `INC A` can be expressed propositionally, where $\boldsymbol{a}[i]$ denotes the i-th bit of $\boldsymbol{a}$ and $\oplus$ denotes the exclusive-or:

$$\texttt{INC A} := \bigwedge_{i=0}^{7} \left(\boldsymbol{a}'[i] \leftrightarrow \boldsymbol{a}[i] \oplus \bigwedge_{j=0}^{i-1} \boldsymbol{a}[j] \right)$$

Similar encodings can be derived for the entire instruction set [5]. The value of these encodings is that optimal transfer functions for either single instructions or whole sequences of instructions can be derived using successive calls to a decision procedure, in this case a SAT solver, prior to executing the actual analysis. Affine equalities [15] are systems of the form $\bigwedge_{i=0}^{m-1}(\sum_{j=0}^{n_i-1} \lambda_{i,j} \cdot v_j = d_i)$, where v_j are program variables and $\lambda_{i,j}, d_i \in \mathbb{Z}$, which can be used to describe relations between variables. Our approach derives optimal affine transformers for basic blocks from the Boolean encodings, using the algorithm developed by Brauer and King [4, Sect. 3.2]. As an example, consider the above instruction, and for brevity, let $\langle\!\langle \boldsymbol{a} \rangle\!\rangle = \sum_{i=0}^{7} 2^i \boldsymbol{a}[i]$. Then, we obtain the following affine system:

$$(\langle\!\langle \boldsymbol{a} \rangle\!\rangle \leq 254) \Rightarrow (\langle\!\langle \boldsymbol{a}' \rangle\!\rangle = \langle\!\langle \boldsymbol{a} \rangle\!\rangle + 1) \qquad (\langle\!\langle \boldsymbol{a} \rangle\!\rangle = 255) \Rightarrow (\langle\!\langle \boldsymbol{a}' \rangle\!\rangle = 0)$$

Using this representation, linear constraints — most notably octagons [17] — that distinguish inputs that lead to overflows are derived from the Boolean formulas. Otherwise, no affine relation between $\boldsymbol{a}$ and $\boldsymbol{a}'$ could be determined since, e.g., $254 + 1 = 255$ and $255 + 1 = 0$ in unsigned machine-arithmetic.

2.2.2 Local invariants through interval analysis

Interval analysis determines invariants using the computationally attractive interval abstract domain [8]. Let $\mathbb{N}^\star = \{0, \ldots, 255\}$ denote the set of numbers representable with a single 8-bit word. Then, a word-level interval is composed of $[a, b]$ with $a, b \in \mathbb{N}^\star$ and $a \leq b$. With $\top = [0, 255]$, $\bot = \emptyset$, and a *join* defined as $[a_1, b_1] \sqcup [a_2, b_2] = [min(a_1, a_2), max(b_1, b_2)]$, the domain forms a complete lattice.

To illustrate interval arithmetic, consider an `ADD A,B` instruction, summing the operands `A` with `B` and storing the result back to `A`. Suppose, we enter the instruction with the intervals $\texttt{A} = [12, 74]$ and $\texttt{B} = [10, 14]$, then we can derive that the resulting value in `A` will be within the interval $[12 + 10, 74 + 14] = [22, 88]$. These invariants are derived for each program counter location using fixed-point iteration and a combination with affine relations, following the reduction algorithm described in [5, Sect. 6]. More details are given in [6].

As a result, the analysis yields a list of word-level intervals over memory locations attached to every *pc* location, i.e., $\langle pc, (\texttt{A}[a_0, b_0], \texttt{B}[a_1, b_1], \ldots) \rangle$. These invariants are used to detect potential violations of the specification. For example, if the global invariant $\mathcal{I}(\texttt{A} < 25)$ should hold, then we identify all locations as potential violations that have intervals for `A` including valuations ≥ 25. The test-trace generation algorithm starts from these program locations.

2.3 Test-trace generation

Our algorithm starts from a program location where the specification may be violated, and systematically searches for traces that lead to this violation. Given an assertion Ψ and an invariant θ, we convert $\neg\Psi$ into a disjunctive normal form and treat $\neg\Psi \wedge \theta$ as the desired postcondition. Next, we apply the affine transfer function in reverse using *integer linear programming*, which gives us a precondition, and then, this step is iteratively applied for all possible predecessors, until the entry of the program is reached. The preconditions are computed in breadth-first order, which guarantees that shortest paths to the entry are found. For reasons of continuity, we defer the presentation of an example to Sect. 3.

2.4 Test-trace deployment and execution

A single test trace t is a path of program counter locations $\pi := \langle pc_0, \ldots, pc_n \rangle$ with $pc_i \in \mathsf{Locs}$ and a set of external inputs $In := \langle pc, i \rangle$ attached to certain program locations. For example, $In := \langle 0xC1C1, \texttt{p1} \leftarrow 0xB2 \rangle$ represents that $0xB2$ will be provided on I/O port `p1` at program counter location $0xC1C1$.

In our approach, we do not explicitly alter the code itself, nor do we insert additional event-triggers into the source code, which is a common practice in runtime verification [12]. Our monitoring is done by a hardware monitor unit, attached to an industrial IP core of the target microcontroller. The whole execution takes place on an FPGA, connected to the actual environment of the application. The monitor unit allows us to non-intrusively and on-the-fly monitor and track memory accesses of the microcontroller core. Besides, the monitor compares the current program counter with the expected one given in π. Whenever this comparison fails, we halt the microcontroller, mark t as *infeasible,* and load the next test trace, thus, subsequently ruling out spurious test traces. However, if the unexpected branch was caused by an indirect jump, we add the newly detected jump target to the CFG. In case the actual execution follows the predicted path π, the monitor will verify whether the specification items hold along the path (for global invariants) or on certain program locations (for local assertions).

3 Worked Example

Fig. 2 shows an embedded C code snippet and its CFG. The labels of the CFG nodes relate to the program counter locations on the left. The code reads a sensor value from an 8-bit input port and converts the value to its absolute value, storing the result in *val*. Next, a while loop is entered sending *val* times PWM pulses to the output and decrementing *val* each iteration. Whenever the predicate $val > 0$ is violated the assertion is reached and the loop starts again.

Based on a first intuition, the assertion will hold, regardless of the sensor values. The presumably positive variable *val* is decremented towards 0. Interestingly, the assertion does not hold under all inputs. Consider the binary sensor input $b1000000$, which corresponds to -128 in two's complement. The `ABS` macro will not alter the value since $-(-128) = -128$ due to the limited bit-width. It is obvious that the predicate $(-128 > 0)$ at the beginning of the while loop is false and the assertion does not hold.

Our algorithm starts by negating the predicate in the assertion, which gives $(\texttt{val} < 0) \vee (\texttt{val} > 0)$ in program location 6. The assertion has a single predecessor, i.e., node 3, for

```
#define ABS(a) (((a)<0)?-(a):(a))
char getSensor(void){return P1;}
void sendPWM(void){...;}

    int main(void){
      char val;
(1)   while(1){
(2)     val = ABS(getSensor());
(3)     while(val > 0){
(4)       sendPWM();
(5)       val--;
        }
(6)     ASSERT(val == 0);
      }
    }
```

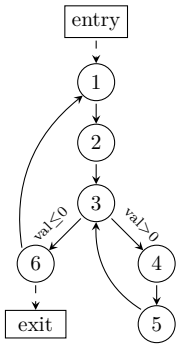


Figure 2 Example code (left) and CFG (right)

which we have derived the following transfer function:

$$\begin{aligned}
&(\texttt{getSensor()} \geq 0 \wedge \texttt{getSensor()} \leq 127) && \Rightarrow (\texttt{val}' = \texttt{getSensor()})\\
&(\texttt{getSensor()} \geq -127 \wedge \texttt{getSensor()} \leq -1) && \Rightarrow (\texttt{val}' = -\texttt{getSensor()})\\
&(\texttt{getSensor()} \geq -128 \wedge \texttt{getSensor()} \leq -128) && \Rightarrow (\texttt{val}' = -128)
\end{aligned}$$

The third one is examined, which gives us a test trace with inputs that lead to a violation of the assertion, namely $\pi = \langle 1, 2, 3, 6\rangle$; $In = \langle 2, \texttt{getSensor()} \leftarrow -128\rangle$ where the input in line 2 is -128. This test trace is executed on the IP core and the runtime monitor confirms that π is indeed a real counterexample trace.

4 Related Work

Test-case generation using formal methods, is an active area of research. Cousot and Cousot introduce abstract interpretation based program testing as *abstract testing* in [9], an approach closely related to our work. However, we apply abstract interpretation to machine code and offer a way to automatically rule out spurious counterexamples. Another popular approach is to use model checkers to derive test suites that comply with industrial coverage criteria [11]. With increasing complexity, these approaches suffer from similar problems as traditional model checking.

Wenzel et al. [22] describe cross-platform verification of embedded C code. Platform-specific C code is translated into semantically equivalent C code used by CBMC to generate counterexamples, which are executed on the host and on the target platform. Thus, their approach can find errors introduced by the compiler. Our approach is independent of the high-level implementation and does not require to instrument the code, which is vital for verifying timing properties. Deriving test data for machine code with a structural coverage goal is described in [2]. Their tool Osmose translates executable code to a generic assembly language and uses concolic execution for path exploration.

5 Discussion & Future Work

5.1 Summary

In this paper, we have addressed the question of deriving test cases from microcontroller binary code. Unlike other techniques, our approach uses abstract interpretation using a combination of different abstract domains to derive test cases directly from the executable

program code. The purpose of our work is not necessarily to derive test cases that satisfy certain coverage criteria, but rather to systematically infer paths that exhibit faulty behavior.

5.2 Future Work

In addition to the global and local assertions (cf. Sect. 2.1), we want to include time-bounded properties of the form $\Theta(\varphi_1, \varphi_2, \delta)$. Such properties state that if the predicate φ_1 holds then φ_2 must hold within $\delta \in \mathbb{N}$ clock cycles. Clearly, future efforts also include a case study showing the feasibility of our approach when applied to industrial embedded code.

Acknowledgements The work of Thomas Reinbacher and Andreas Steininger has been supported within the FIT-IT project CEVTES managed by the Austrian Research Agency FFG under grant 825891. The work of Martin Horauer has been supported within the FHplus project DECS managed by the Austrian Research Agency FFG under grant 811414. The work of Jörg Brauer and Stefan Kowalewski has been, in part, supported by the UMIC Research Centre of Excellence at the RWTH Aachen University.

References

1 G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What you see is not what you execute. In *VSTTE*, Toronto, Canada, 2005.

2 S. Bardin and S. Herrmann. OSMOSE: Automatic structural testing of executables. *Softw. Test., Verif. & Reliab.*, 2009.

3 A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

4 J. Brauer and A. King. Automatic abstraction for intervals using boolean formulae. In *SAS*, volume 6337 of *LNCS*, pages 167–183. Springer, 2010.

5 J. Brauer, A. King, and S. Kowalewski. Range analysis of microcontroller code using bit-level congruences. In *FMICS*, volume 6371 of *LNCS*, pages 82–98. Springer, 2010.

6 J. Brauer, T. Noll, and B. Schlich. Interval analysis of microcontroller code using abstract interpretation of hardware and software. In *SCOPES*. ACM, 2010.

7 E. M. Clarke and H Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 41–43. Springer, 2004.

8 P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd International Symposium on Programming*, pages 106–130, April 1976.

9 P. Cousot and R. Cousot. Abstract interpretation based program testing. In *SSGRR*. Scuola Superiore G. Reiss Romoli, 2000. Invited paper.

10 E. Eide and J. Regehr. Volatiles are miscompiled, and what to do about it. In *EMSOFT*, pages 255–264. ACM, 2008.

11 G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers: a survey. *Softw. Test., Verif. & Reliab.*, 19(3):215–261, 2009.

12 K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Form. Methods Syst. Des.*, 24(2):189–215, 2004.

13 M. P. E. Heimdahl, D. George, and R. Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In *HASE*, pages 178–186. IEEE, 2004.

14 C.A.R. Hoare. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25:14–25, 2003.

15 M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

16 J. Kinder, H. Veith, and F. Zuleger. An abstract interpretation-based framework for control flow reconstruction from binaries. In *VMCAI*, volume 5403 of *LNCS*, pages 214–228. Springer, 2009.

17 A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

18 T. Reinbacher, M. Horauer, B. Schlich, J. Brauer, and F. Scheuer. Model checking assembly code of an industrial knitting machine. In *EM-Com*, pages 97–104. IEEE, 2009.

19 T.W. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, volume 2937 of *LNCS*, pages 252–266. Springer, 2004.

20 RTCA/DO-178B. Software considerations in airborne systems and equipment certification, 1992. Washington DC, USA.

21 B. Schlich. *Model Checking of Software for Microcontrollers*. Dissertation, RWTH Aachen University, Aachen, Germany, June 2008.

22 I. Wenzel, R. Kirner, B. Rieder, and P. Puschner. Cross-platform verification framework for embedded systems. In *SEUS*, pages 137–148. Springer, 2007.

Instructor Selector Generation from Architecture Description

Miloslav Trmač[1], Adam Husár[1], Jan Hranáč[2], Tomáš Hruška[1], and Karel Masařík[1]

1 **Brno University of Technology, Faculty of Information Technology**
{itrmac, ihusar, hruska, masarik}@fit.vutbr.cz
2 **ApS Brno, s.r.o.**
hranac@aps-brno.cz

Abstract

We describe an automated way to generate data for a practical LLVM instruction selector based on machine-generated description of the target architecture at register transfer level.

The generated instruction selector can handle arbitrarily complex machine instructions with no internal control flow, and can automatically find and take advantage of arithmetic properties of an instructions, specialized pseudo-registers and special cases of immediate operands.

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.109

1 Introduction

Application-specific processors are often used in embedded applications with large production quantities due to the speed and low power consumption they can provide at modest cost compared to using a generic CPU with higher execution speed. The Lissom project[8] developed at the Brno University of Technology aims to provide a full development environment for iterative hardware-software codesign, allowing embedded system developers to rapidly experiment with application-specific architecture facilities by automatically generating software development tools (assembler, disassembler, linker, simulator, C compiler) and a hardware prototype. Typically, the embedded system developer would experiment with changing the CPU, e.g. adding specialized instructions, and evaluate each experiment by using the Lissom software to regenerate the tool chain, recompile the application, and test its performance on a simulator.

This effort includes automatic generation of a C compiler. We have decided to base this work on the open-source LLVM project[6], reusing its existing front-end (which converts input in C into the LLVM internal representation), its optimization passes, and the provided infrastructure for implementing back-ends (which convert the internal representation into assembler or binary code).

In the compiler front-end it is only necessary to provide information about the application binary interface (ABI) of the target platform; front-ends are already prepared for this, so we only need to describe data types in a predetermined format. The LLVM optimization passes work purely on the internal representation without considering the target architecture. Most of the work therefore involves the compiler back-end, which is described in this article, focusing primarily on the instruction selection component.

2 Related Work

A full-featured compiler generator is described in [5]: from an architecture description in a language called LISA, it generates input for the commercial CoSy compiler development

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 109–115

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

system. LISA is suitable primarily for accurate simulation, so compiler generation is not automatic: register properties and instruction scheduling information is extracted automatically, but the user must manually provide instruction behavior descriptions using a provided GUI. If the LISA input is changed, the information in the GUI may become obsolete and needs to be updated manually. In contrast, in our approach the instruction selector is generated automatically from the same input file used for generating other tools, shortening the edit-compile-evaluate cycle of architecture design exploration.

An approach that does not ask the user to specify the instruction selector described in [2]. The necessary instruction operation descriptions are extracted from an architecture description automatically. After phases that simplify the instruction behavior descriptions, the descriptions are matched against manually prepared patterns to find instructions that can be used in the initial instruction selection pass of `gcc` (`gcc` does not use a tree pattern selector, and requires named templates for the atomic operations). Unlike our approach, instructions that combine several basic operations are not supported.

An entirely different approach is not to try to support arbitrary user-specified architectures. Such systems are based on a specific CPU architecture, and provide a combined hardware/software compiler to the user. Given an input program, the compiler produces description of hardware (e.g. in VHDL) that implements some parts of the program, and executable code for the base architecture augmented with the described hardware, that implements the rest.[9] Such systems can produce good results if the base architecture is suitable, but are difficult to adapt when the base architecture needs to be replaced.

3 ISAC and Instruction Semantics Extraction

ISAC is an architecture description language (ADL) based originally on LISA.[7] It is a mixed ADL, meaning that it allows to describe both architecture and microarchitecture.

The architectural description consists of registers, memories, and instruction set description. Registers and memories are described as global C language variables. Instruction set is described hierarchically, because most instructions can use the same register or immediate operands. Two main constructs are used to describe the instruction set: The `OPERATION` construct allows to describe parts of instruction's syntax, binary encoding, and semantics. Instances of other operations or groups may be used in an operation specification. `GROUP` is used to describe situations where an instance used in an operation can be one of a set of operations or groups. An example of a "store byte" instruction is in listing 1.

Listing 1 Example of a "store byte" instruction with indirect adressing mode

```
# This is an simplified example, an operation usually represents
# multiple instructions (e.g. also "store half-word", and
# "store word").
OPERATION instr_direct_loadstore {
  INSTANCE register ALIAS {rt, base};
  INSTANCE signed_imm16 ALIAS {offset};
  ASSEMBLER { "SB" rt "," offset "(" base ")" };    # Syntax
  CODING { 0b100100 base rt offset };               # Binary encoding
  BEHAVIOR {                          # Instruction behavior
    int addr = regs[base] + offset;   # regs is an array of
    char val = regs[rt] & 0xFF;       #   general-purpose registers
    mem[addr] = val;            # mem represents memory address space
  };
}
```

The compiler back-end needs to identify each particular target instruction, but there is no notion of an instruction in the ISAC language. What can be used here, is that the assembly syntax description is based on context-free grammars. If we generate all words of the assembly language, we get a list of instructions. For each instruction the corresponding behavior in C is generated as well. This C code representing instruction semantics is then converted to a common format, a few simple optimizations that simplify the semantics are performed, and this result is passed to the back-end generator. The result for the example "store byte" instruction can be seen in listing 2.

Listing 2 Instruction semantics example, corresponding to listing 1

```
# Temporary variable names were changed to make the example easier to
# follow.
instr instr_direct_loadstore__op_sb__gpr_std__simm16__gpr_std__,
  %imm = i16 immop(1);
  %Rx = i32 regop(cl0, 0);
  %Rx_trunc = trunc(%Rx, i32 8);
  %imm_ext = sext(%imm, i32 32);
  %Ry = i32 regop(cl0, 2);
  %addr = add(%Ry, %imm_ext);
  store(%Rx_trunc, %addr);
, "SB" 0 "," 1 "(" 2 ")", 1  # Assembler syntax
```

4 Instruction Selector Generation

Instruction selection is the largest component of a LLVM compiler back-end. Its purpose is to convert an input program from a target-independent internal representation into a lower-level representation that deals with instructions of the target architecture instead of generic operations.

4.1 Instruction Semantics Format

The primary result of the instruction semantics extraction process described in section 3 is a list of instructions. In contrast to the human concept of a "single instruction", where all binary formats that use the same assembler mnemonics are considered a single instruction, we define a single instruction as a maximal set of binary encodings within which semantics and binary encoding can change only by substituting one register by another register from the same register class, or one constant by another constant of the same bit width and format.

For example, `load R1 = [R1 + R2]` and `load [R1 + imm]` are different instructions although they share the "load" mnemonics. Also, `add R1 = R2 + R3` and `add R1 = R2 + R0` (where `R0` denotes a read-only pseudo-register with zero value) are different instructions: they share the same binary format, but the semantics of one is "add values of two registers", and the semantics of the other is "copy a register value." In contrast, `cmov if(R1) R2 = R3` is (necessarily) considered a single instruction, where the semantics depends on the value of a register, not on its identity.

The extracted semantics is a sequence of *atomic operations*, using an unbounded number of temporary variables. Only limited control flow is supported: an `if` operation can be used to delineate a conditionally-executed set of atomic operations, but control flow can not return to the "main path" after executing the `if` body. Loops are not supported, so the control flow graph can form at most a tree rooted at entry of the semantics. In this control

flow format, it is easy to arrange temporary variables to form the industry-standard SSA representation[4]. In contrast, the semantics can modify a single physical register repeatedly, so physical registers are not in SSA form.

4.2 Basic LLVM Instruction Selector

LLVM uses a tree pattern matching instruction selector, which can take advantage of complex instructions, as long as they have only one result. In contrast to research in this area, which suggests using bottom-up analysis with dynamic programming and generating globally optimal instruction selections (within the assumed cost model)[1, 3], LLVM uses a top-down, only locally optimal selector, sacrificing code quality for flexibility and speed. The selector is automatically generated from instruction descriptions, but it also allows adding additional C++ code to handle more complex cases.

Description of each instruction in LLVM includes its assembler format, (variable) operands, effects on other (fixed) registers, and other information, e.g. flags describing important behavior, and optionally binary format of the instruction. Instructions that should be handled by the generic instruction selector must also include semantics description in an expression tree form.

Instructions for which the extracted semantics naturally describe an expression tree (single externally-observable output—either register assignment or a side effect, no conditionals) can be converted to the LLVM format by treating the use-def links in the linear instruction semantics description as parent-child links in an expression tree. Instructions that change a register value conditionally can be converted into a corresponding LLVM operation as well, by implicitly constructing the C "?:" operator within the semantics.

An example LLVM instruction description is provided in listing 3.

Listing 3 LLVM instruction description example, corresponding to listing 2

```
def instr_direct_loadstore__op_sb__gpr_std__simm16__gpr_std__:
  LissomInst
  <(outs), (ins cl0:$op0, cl0:$op2, Si32i16imm:$op1),     # Operands
   "SB $op0 , $op1 ( $op2 )",                             # Assembler
   [(truncstorei8 cl0:$op0,                               # Expression
                  (add cl0:$op2, (i32 sextimm16:$op1)))]>{}
```

This approach can handle a large number of instructions, including instructions that combine several operations. On the other hand, this alone is insufficient on almost all architectures, because many architectures do not provide some of the "atomic" operations in a pure form, and if the "atomic" operations are not available to the instruction selector, there are likely to be programs that cannot be compiled.

4.3 Instructions with Multiple Outputs

Many architectures support instructions that provide more than one output (in this section, "output" means storing a value in a register, or a side effect). These instructions are often primarily used for only one of the outputs. This includes almost all instructions on architectures that use a flags register. To handle these cases, any instruction that has more than one output is *cloned*. One clone is created for each output, and in each clone, the other outputs are made invisible to LLVM (setting of a register is replaced by an annotation that the register is clobbered by an indeterminate value). If one of the outputs can not safely be made invisible (e.g. a jump), the clone is discarded. Thus, a single `xor Rx = Ry ^ Rz`

instruction on an architecture with a flags register is cloned into `xor_reg`, which sets `Rx` and clobbers the flags register, and `xor_flags`, which sets the flags register and clobbers `Rx`. The LLVM instruction selector can use the `xor_reg` clone and thus automatically take advantage of the instruction.

4.4 Specialized Instruction Outputs

On many architectures some atomic operations are available only as a part of a more generic instruction. For example, an architecture might provide only a single flexible `load` instruction: `load Rx = [Ry + Rz * imm1 + imm2]` (where values of `imm1` and `imm2` are often limited in range). By choosing `imm1 = 1`, `imm2 = 0`, or `imm1 = imm2 = 0`, we can get the simpler `load Rx = [Ry + Rz]`, and `load Rx = [Ry]`, respectively. To handle these cases, we attempt to specialize each instruction with chosen constant values.

First, promising values of immediate operands are collected: Basic dominator optimizations (dead code deletion, copy propagation, constant folding, arithmetic simplification) are performed on the semantics, and each atomic operation that refers to an immediate operand is examined for values of the operand that could allow constant folding the operation. For example, values `0` and `1` are used for multiplication operands, or values `0` and `~0` (of correct width) are used for operands of bit-wise operations. Other atomic operations, including other arithmetic operations, comparisons, and `if`, are handled similarly. Finally, the value `0` is always added, simply because it so often leads to simplification.

After all candidate values are collected for each immediate operand, the instruction is cloned: one clone is created for each possible assignment of candidate values to respective immediate operands (including the cases when a specific value is not assigned to some of the operands). Each clone is then re-optimized: if the optimization does not simplify the instruction semantics, the clone is discarded. The remaining clones are treated exactly the same as "native" instructions (e.g. an LLVM description is generated for them).

4.5 The Instruction Set as a Whole

In addition to cooperating with the LLVM instruction selector, LLVM needs some information about the overall structure of the instruction set.

Most important is the LLVM "legalization" pass, whose purpose is to modify the input program to only use operations that are available in the target architecture, e.g. converting a 64-bit multiplication into a sequence of 32-bit multiplications and additions. Unfortunately LLVM is not able to extract the required information about available instructions from the individual instruction descriptions, so this information has to be generated separately.

Second, the LLVM instruction selector build process can not handle instructions sets in which two or more instructions match the same expression subtree; the backend author must explicitly select the instruction that the instruction selector will use for the subtree. (The other instructions can still be made available to LLVM—and used perhaps through specialized built-in functions—but their description must not contain the expression subtree, making them unavailable to the instruction selector.) To do this, structurally identical expression subtrees are identified in the backend generator, and a single instruction is chosen from each set of duplicates.

Finally, LLVM needs to generate some target instructions after instruction selection has finished, notably instructions for moves and memory accesses necessary for register allocation and spilling. These instructions are located by looking for instructions matching a specific form of atomic operations that do not have any unwanted side effects.

5 Other Backend Tasks

In addition to instruction selection, a LLVM backend needs to provide information to the register allocator, mainly description of register classes and lists of registers unavailable for allocation. This information is already provided in the extracted instruction semantics, based primarily on user's annotations in the source ISAC file.

We do not currently extract enough information about the pipeline and usage of functional units by instructions to implement effective instruction scheduling.

Finally, a back-end must implement handling of function frames, function calls, parameter passing, and other ABI-dependent transformations. We handle this by looking for instructions matching a specific form of atomic operations, similar to the case of target instructions used after instruction selection. The specific ABI can not be automatically determined, and means to allow the user to specify it are currently being added to ISAC. Without such information, the backend generator automatically generates a reasonable ABI by examining the existing instructions (e.g. looking for "return" or "call"), or, on very regular architectures with little built-in function call support, by looking for indirect addressing modes suitable for managing a stack manually.

6 Experimental Results

We have used the back-end generator on a restricted model of the MIPS architecture developed for the purpose. The semantics extraction implementation resulted in a description of 139 individual instructions (using the definition of "instruction" given in section 4.1). Out of these 139 instructions, 62 could be directly used by the LLVM instruction selector, remaining instructions required additional handling and conversion. Because the architecture includes a `R0` pseudo-register equal to a immediate value of 0, the instruction extraction process recognized a large number of instruction variants involving the `R0` register that ultimately resulted in the same LLVM semantics: in particular, semantics of 17 instructions was `Rx = 0`, and semantics of 11 instructions was `Rx = Ry`.

In our model, the only cloned instructions with multiple outputs were variants of division and multiplication.

Specializing instructions using specific values of immediate operands based on the original 139 instructions resulted in 26 new instruction clones, with 17 distinct classes of semantics for the purpose of the LLVM instruction selector. Similarly to the handling of the `R0` register in semantics extraction, the most frequent specializations resulted in simple register-to register copy and register assignment, but this process also identified ways to provide the primitive load/store operations (e.g. `LW Rx, 0(Ry)` - load a 16-bit value from address given by register `Ry` to register `Rx`), which are necessary both to guarantee ability of the instruction selector to handle arbitrary programs, and to implement register spilling in the register allocator.

7 Conclusion and Future Work

In this article we have presented primarily our approach to automatic generation of a LLVM instruction selector. While the other components of the LLVM backend generator are sufficient to create a working MIPS backend, implementation experience with more architectures is necessary before we can be confident in the viability of our approach and before we can present it in detail.

Our goals for future work include extracting enough information for instruction scheduling, which will also allow choosing the best possible instruction for the instruction selector when

there are several alternatives. Extensions of LLVM to take advantage of SIMD instructions and WLIW architectures are already under development. We also intend to test performance of code created by the generated backend using industry-relevant benchmarks, and improve our backend generator based on detailed analysis of the compiled code for these benchmarks.

Acknowledgements This research was supported by the grants of MPO Czech Republic FR-TI1/038 and by the Research Plan MSM No. 0021630528. We would also like to thank the anonymous reviewers for their valuable comments.

References

1 Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions of Programming Languages and Systems*, 11(4):491–516, October 1989.

2 Soubhik Bhattacharya. Generation of gcc backend from Sim-nML processor description. Master's thesis, Indian Institute of Technology, Kapur, July 2001.

3 Todd A. Proebsting Christopher W. Fraser, David R. Hanson. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

4 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions of Programming Languages and Systems*, 13(4):451–490, October 1991.

5 Manuel Hohenauer, Hanno Scharwaechter, Kingshuk Karuri, Oliver Wahlen, Tim Kogel, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Gunnar Braun. A methodology and tool suite for C compiler generation from ADL processor models. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1530–1591, 2004.

6 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

7 Roman Lukáš, Tomáš Hruška, Dušan Kolář, and Karel Masařík. Two-way deterministic translation and its usage in practice. In *Proceedings of 8th Spring International Conference - ISIM'05*, pages 101–107, 2005.

8 Karel Masařík, Tomáš Hruška, and Dušan Kolář. Language and development environment for microprocessor design of embedded systems. In *Proceedings of IFAC Workshop on Programmable Devices and Embedded Systems PDeS 2006*, pages 120–125. Faculty of Electrical Engineering and Communication BUT, 2006.

9 Tim Sander, Aditya Vishnubhotla Vijay, and Sorin A. Huss. HW/SW codesignflow with LLVM. 2008 LLVM Developers' Meeting, August 2008.

Integer Programming for Media Streams Planning Problem*

Pavel Troubil and Hana Rudová

Faculty of Informatics, Masaryk University
Botanická 68a, 602 00 Brno, Czech Republic
pavel@ics.muni.cz, hanka@fi.muni.cz

Abstract

Continually increasing demands for high-quality videoconferencing have brought a problem of fully automated environment setup. A media streams planning problem forms an important part of this issue. As the multimedia streams are extremely bandwidth-demanding, their transmission has to be planned with respect to available capacities of network links and the plan also needs to be optimal in terms of data transfer latencies. This paper presents an integer programming solution of the problem and its implementation. The implementation achieved very promising results in performance-evaluating measurements. Compared to previous constraint-based solver, it is capable of finding optimal solution significantly faster, allowing for real-time planning of larger problem instances.

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.116

1 Introduction

Modern computer networks allowing high-bandwidth transmissions have become more and more widespread recently. Their increasing availability heavily supports deployment and user-adoption of advanced collaborative environments. These environments frequently require transmission of very high-bandwidth data streams. Smoothness and enjoyability of synchronous remote collaboration also crucially depends on a low-latency transmission of the data streams.

Setting up an advanced collaborative environment might be a difficult and tedious task, probably undesirably hard for end-users. The setup often comprises configuring of potentially high number of individual components (e. g., data producers, processors, distributors, or consumers), and also data distribution paths in a network. As a bandwidth needed for transmission of the data streams is frequently close to capacities of state-of-the-art backbone links, finding out correct and latency-minimal distribution paths also becomes a very complicated task.

In order to automate the process of the environment setup, the CoUniverse framework has been proposed in [5]. The problem of deciding the data distribution paths has been formally defined as a media streams planning problem (MSPP) in [3]. The MSPP is a network optimization problem close to a multicommodity network flows problem [1]. A survey of network optimization problems can be found in [9]. The MSPP is also strongly related to a multicast routing problem [6] for multiple multicast groups (called multicast packing problem). Unfortunately the multicast service is not proper for our purpose since it is not continuously deployed over the whole Internet, and lacks performance needed for high-speed transmissions. Still methodologies applied to solve this problem can provide an inspiration

* This research is supported by the Ministry of Education, Youth and Sports of the Czech Republic under the project 0021622419.

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 116–123

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

for our work. Heuristics for this problem were presented in [2, 10, 8] and an optimal solution using linear programming is known for single multicast problem even if data streams are allowed to split [4].

A constraint-programming based solver of the MSPP has been implemented in the CoUniverse [3]. This solver is only capable of solving medium-sized instances of the problem quickly enough, i. e., in a few seconds. With the aim to solve larger problem instances, we propose a new solver based on integer programming. We have rewritten the previous constraint programming model to a form of an integer programme, reformulating some improper disjunctive constraints. The IP model was implemented in the CoUniverse framework and evaluated in terms of performance. Results of the evaluation show that our solver is capable of solving larger instances of the problem for real-world network topologies, e. g., those used for distributed lectures.

2 Problem description

Generally said, the media streams planning problem is a network optimization problem of computing data distribution paths in a network. Given a topology of the network, a set of data sources, and destinations for the data delivery, the goal is to find a distribution tree for each of the data sources. The data distribution paths are required to be optimal in terms of overall transmission latency. Media streaming network applications usually require all data transmissions to occur at the same time, e. g., in a videoconference. Hence no temporal parameters are considered in the MSPP. In order to handle changes in the network (e. g., link outage), the CoUniverse monitors the environment and may invoke a replanning, i. e., a new call of the solver.

Main entity of the network is a *node* $v \in V$. Since we abstract from physical network devices such as switches or routers, the nodes represent computers (servers as well workstations), each of them running one or possibly more *applications*. The applications either produce a data stream (called *producers*, $p \in P$), consume the stream (*consumers*, $c \in C$), or distribute it (*distributors*, $d \in D$), possibly creating multiple copies of received data. The applications are capable of processing one data stream at most, i. e., a single application can neither produce, consume, nor distribute more than one stream. If an application $a \in P \cup D \cup C$ is running on a node v, we write $a \in v$. The distributors are server applications providing a multicast functionality on an application layer of the ISO/OSI model. UDP packet reflectors and Active Elements are examples of such applications.

If there is a distributor running on a network node, no other application is allowed to run there, i. e., no producers and/or consumers may run on that node, neither may any other distributor. If there is no distributor on a node, there may be running several producers and consumers together. Their number is not limited as long as they all process different streams. Yet, this model targets primarily on CPU-intensive multimedia applications that might cause overload of the node when run simultaneously. Nodes are organized into *sites*, which generally represent geographical collocation of the nodes. There are typically several nodes at a site, each of them running a single application.

Each network node has several network *interfaces* $i \in I$ configured. We denote by $i \in v$ that the interface i is configured on the node v. An interface has a limited transmission capacity $capI(i)$. Every interface belongs to a *subnetwork*. We consider each interface reachable from all other interfaces in the same subnetwork and vice versa.

In other words, there is a *link* $l \in L$ between each ordered pair of interfaces, i. e., it is strictly directional. Links together with nodes form a directed graph $N = (V, L)$

corresponding to the underlying network. The links represent a network infrastructure which facilitates the data transfer between the pair of network nodes. Although links formally interconnect interfaces, we often speak about them as connecting *nodes*. Naturally, a link between interfaces (i_1, i_2) connects nodes (v_1, v_2) if and only if $i_1 \in v_1 \ \wedge \ i_2 \in v_2$. Source and target nodes of a link l are denoted $\mathit{begin}(l)$ and $\mathit{end}(l)$, respectively. We also denote a set of links connected to an interface i as $\mathit{links}(i)$.

For an *application* $a \in P \cup D \cup C$ on a *node* v, we define $\mathit{inlinks}(a)$ as a set of links ending in the *node* v, i. e., $l \in \mathit{inlinks}(a) \iff a \in \mathit{end}(l)$. We extend this notation on a set of applications $A \subseteq P \cup D \cup C$: $l \in \mathit{inlinks}(A) \iff a \in \mathit{end}(l)$ for some $a \in A$. We also define $\mathit{outlinks}(a)$ and $\mathit{outlinks}(A)$ analogously.

Each network link l has two attributes: its maximum capacity $\mathit{cap}(l)$ and transfer delay $\mathit{latency}(l)$. Since the links do not represent physical topology of a network, several links in our model might share one physical network link. Consequently, whole bandwidth of $\mathit{cap}(l)$ might not be actually available for transmission. A static configuration of *capacity* of the links is augmented by a real-time monitoring in the CoUniverse, similarly as *latency* of the links needs to be also monitored.

The goal of media streams planning is to determine paths for the distribution of the data *streams* ($s \in S$). We use the term *stream* since motivation for the MSPP lies in continuous multimedia transmissions. We denote a bandwidth required for transmission of the stream s as $\mathit{bw}(s)$. Each stream s is produced by a single application $\mathit{producer}(s) \in P$ and is required to be delivered to a set of consumers $\mathit{consumers}(s) \subseteq C$. There is exactly one producer of the stream s and at least one consumer of the stream, i. e., $\mathit{consumers}(s) \neq \emptyset$. Transmission of the streams and possibly creation of multiple data copies is performed by the distributors. More precisely, each stream is transferred from a producer to a set of consumers by a communication tree with the producer in its root, consumers at leafs, and media distributors at internal nodes. Therefore, the problem may be considered as a tree placement [3] in contrast to classical path placement [9] where no data multiplication is processed. On the other hand, all packets of each stream from its producer to a single consumer have to be transferred along the same path. If the packets would be transferred along more than one path, unfavourable reordering of the packets would occur due to different latencies of the paths. The distributors are therefore not allowed to send data between any producer — consumer pair from a node through more than one link (e. g., for load balancing purposes).

3 Integer Programming Model

For each stream s and network link l, we introduce a binary decision variable $x_{s,l}$, further denoted as *streamlink*. The streamlink $x_{s,l}$ equals to 1 if the stream s is transmitted over the link l, otherwise it corresponds to 0.

We also call a streamlink $x_{s,l}$ *active* if and only if $x_{s,l} = 1$, or *inactive* otherwise. Since our aim is to minimize overall transmission latency, we formulate the objective function as a sum of latencies of all active streamlinks.

$$\min \sum_{s \in S} \sum_{l \in L} x_{s,l} \cdot \mathit{latency}(l)$$

The three following constraints implement network capacity limitations. By constraint (1), it is not allowed to transfer a stream s through a link l of insufficient capacity. We set the decision variable directly to zero when the link does not have sufficient capacity for

transmission of the stream. This constraint is a redundant one and follows from the consequent constraint (dependent on decision variables). Constraint (2) guarantees that the capacity of a link l cannot be exceeded by a total bandwidth of the streams transferred over the link. Constraint (3) states that the capacity of an interface i cannot be exceeded by a total bandwidth of the streams transferred over all links connected to this interface. The presented variant of the constraint is used for interfaces which do not support full duplex. On full duplex interfaces, incoming and outgoing links are treated separately, since they do not interfere with each other.

$$x_{s,l} = 0 \qquad \forall s \in S \, \forall \, l \in L \text{ s.t. } bw(s) > cap(l) \tag{1}$$

$$\sum_{s \in S} bw(s) \cdot x_{s,l} \leq cap(l) \qquad \forall l \in L \tag{2}$$

$$\sum_{s \in S} \sum_{l \in links(i)} bw(s) \cdot x_{s,l} \leq capI(i) \qquad \forall i \in I \tag{3}$$

A network node cannot send a stream s over arbitrary outgoing links. To allow transmission of the stream s over a link l, either consumer of this stream or a distributor must reside on the target node of the link l (4). A similar rule holds for receiving of the stream. To allow transmission of the stream s over a link l, either producer of this stream or a distributor must reside on the source node of the link l (see constraint (5)).

$$x_{s,l} = 0 \qquad \forall s \in S \, \forall \, l \notin inlinks(D \cup consumers(s)) \tag{4}$$

$$x_{s,l} = 0 \qquad \forall s \in S \, \forall \, l \notin outlinks(\{producer(s)\} \cup D) \tag{5}$$

Each producer is capable of producing a stream and sending it to another network node and does not have any additional data distribution capabilities. Consequently any producer is required to send the stream over exactly one link.

$$\sum_{l \in outlinks(producer(s))} x_{s,l} = 1 \qquad \forall s \in S \tag{6}$$

If there is more than one consumer of a stream s, its producer is not allowed to send the stream directly to any consumer. We disable all direct links from the producer to all consumers of the stream s. We introduce the constraint (7) for each stream s such that $\|consumers(s)\| > 1$.

$$x_{s,l} = 0 \quad \forall l \in L \quad \text{s.t.} \quad l \in outlinks(producer(s)) \cap inlinks(consumers(s)) \tag{7}$$

This constraint is redundant, since any directlink form $producer(s)$ to any consumer of the stream would leaverequests of the other consumers unsatisfied without breaking the constraint (6).

Each producer is required to send the data to each consumer along a single path. This means that any consumer does not need to receive a stream from more than one node unless there is some redundant transmission. The consumer is therefore required to receive the stream over exactly one link.

$$\sum_{l \in inlinks(c)} x_{s,l} = 1 \qquad \forall s \in S \, \forall c \in consumers(s) \tag{8}$$

The following constraints are aimed to make each stream s transferred along a tree rooted at a node where $producer(s)$ resides. A distributor d is allowed to distribute one

stream at most, and the stream has to be transferred to any network node only by a single network link. In addition, each node containing a distributor cannot contain any other application. Following these rules, there may be at most one active streamlink incoming in the distributor's node (see (9)). Constraints (10) and (11) guarantee that a stream s is sent by a distributor d if and only if it is also received by d. Contrary to the constraint (9), they have to be formulated on a per-stream basis. In this case, summing all streamlinks would allow the distributor to send further arbitrary stream no matter which stream it receives. Since corresponding constraints of the constraint programming model [3] are not suitable for the integer programming due to their improper statement with disjunctions, their reformulation was necessary. Constraint (10) states that there are not less active outgoing links than active incoming links, i. e., the distributor d is forced to forward an incoming stream. Next, a distributor d is not allowed to forward any stream it does not receive. As the distributor may only distribute a single stream, $\|outlinks(d)\|$ corresponds to the maximum possible number of streamlinks over which the distributor may send any data (11).

$$\sum_{s \in S} \sum_{l \in inlinks(d)} x_{s,l} \leq 1 \qquad \forall d \in D \tag{9}$$

$$\sum_{l \in inlinks(d)} x_{s,l} \leq \sum_{l \in outlinks(d)} x_{s,l} \qquad \forall s \in S \ \forall d \in D \tag{10}$$

$$\|outlinks(d)\| \cdot \sum_{l \in inlinks(d)} x_{s,l} \geq \sum_{l \in outlinks(d)} x_{s,l} \qquad \forall s \in S \ \forall d \in D \tag{11}$$

A distribution tree of each stream s is limited in size by the number of consumers of s and the number of available distributors. There are three redundant constraints (12), (13), and (14) to support that. First, the distribution tree may include at most $1 + \|D\| + \|consumers(s)\|$ nodes. Since cycles among nodes are not allowed (see constraint (15)), maximum number of links over which the stream s may be transferred is equal to $\|D\| + \|consumers(s)\|$ (see (12)). Next, if there is only a single consumer of a stream s, one link may be sufficient for transmission (13). Otherwise, the stream has to be transmitted through at least one distributor, setting the minimal number of needed links to $1 + \|consumers(s)\|$ (see (14)).

$$\sum_{l \in L} x_{s,l} \leq \|D\| + \|consumers(s)\| \qquad \forall s \in S \tag{12}$$

$$\sum_{l \in L} x_{s,l} \geq 1 \qquad \forall s \in S \text{ s. t. } \|consumers(s)\| = 1 \tag{13}$$

$$\sum_{l \in L} x_{s,l} \geq 1 + \|consumers(s)\| \qquad \forall s \in S \text{ s. t. } \|consumers(s)\| > 1 \tag{14}$$

The last constraint eliminates cycles that might occur among nodes with distributors. The previous constraints allow existence of a cycle in which each distributor may receive a stream from another distributor, potentially forwarding the stream on a path to one or more consumers. The cycle-avoidance constraints are derived from the graph theory results. If a graph with k vertices has more than $k - 1$ edges, there is a cycle in the graph. Further, if there is not a cycle in any subgraph of a graph then the graph does not contain any cycle either. We denote the number of distributors $\|D\|$ as n. To avoid cycles among the distributors, we put an upper bound on the number of edges in each k-tuple of distributors for $2 \leq k \leq n$. For n distributors, there are $\binom{n}{k}$ subsets of k elements in total, i. e., k-tuples of distributors. We denote a set of all distributor k-tuples as D_k, and its i-th member as $D_k(i)$.

For each stream s and each k-tuple of distributors, we introduce the following constraint.

$$\sum_{\substack{j_1,j_2 \in D_k(i) \wedge \\ v_{j_1}=begin(l) \wedge v_{j_2}=end(l)}} x_{s,l} \leq k-1 \quad \forall s \in S \, \forall k \in \{2,\ldots,n\} \, \forall i \in \{1,\ldots,\tbinom{n}{k}\} \tag{15}$$

This formulation is similar to cycle-avoidance constraints in subtour formulation of cycle elimination in the travelling salesman problem (TSP) [7]. The main difference is that we need to apply the constraint even for cycles containing more than $n/2$ nodes. In the TSP, occurrence of larger cycle would necessarily enforce occurrence of another cycle with less than $n/2$ nodes; yet, this assumption does not hold in the MSPP.

Redundant Constraints

The redundant constraints were kept in the model although they do not strengthen the formulation. On the other hand, they may improve performance of the MIP solvers significantly. Evaluation of their influence on the solver performance will be part of our follow-up work.

4 Evaluation and Results

We modified an MSPP solving module in the CoUniverse to implement the integer programming model. The module is written in the Java programming language and uses the Gurobi Optimizer[1] version 3.0.0 as a backend MIP solver.

Three topologies simulating typical data distribution patterns in advanced collaborative environments were chosen for evaluation of the solver performance:

(a) *1:n* topology: one site *si* transmits a stream to all other sites through a single distributor, and each of the other sites transmits a stream back to *si*. Further denoted *1:n-s*.
(b) *1:n* topology: it is similar to the previous one with an exception of higher number of distributors — there is one for each site except *si*. The topology is further denoted *1:n-r*.
(c) *m:n* topology: each site transmits a stream to all other sites through its own distributor(s).

These topologies were taken from [3] to compare with their results (see this paper for more detailed description of the topologies and their relation to real-world problems).

All measurements were performed on a PC equipped with Intel Xeon 5160 @ 3.0 GHz quadcore processor and 6 GB RAM, running Linux 2.6.22-17 and Java SDK 1.6.0 in a virtualized Xen environment. Options for `java` were set to `-server -da -dsa`. Each measurement was continuously repeated 20 times, and only the last 5 runs were taken into account. A measurement timer had 4 ms resolution. We did not limit the number of processor cores available to the Gurobi optimizer. Unfortunately, we did not observe any significant performance differences when compared to single-thread runs.

Numbers of nodes and links in instances of the topologies (parametrized by number of sites) are shown in Table 1. The number of links is presented after an elimination process (same as the one applied in [3]).

Results of the performance measurements are shown in Table 2. The measured times (in milliseconds) are split in two parts: preparation (creation of variables and constraints, elimination of the links), and optimization, which is performed by the backend solver solely. The largest instances of each topology represent current limitation of the solver for real-time application. In case of the *1:n-s* topology, the preparation phase is the bottleneck. The *1:n-r*-12 topology is already above interactivity requirements. Steep growth in the computation

[1] `http://www.gurobi.com`

Table 1 Parameters of topologies used for performance measurement

Topology	*1:n-s*-2	*1:n-s*-4	*1:n-s*-8	*1:n-s*-16	*1:n-s*-32	
Nodes	5	11	23	47	95	
Edges	10	44	184	752	3,040	
Topology	*1:n-r*-2	*1:n-r*-4	*1:n-r*-6	*1:n-r*-8	*1:n-r*-10	*1:n-r*-12
Nodes	5	13	21	29	37	45
Edges	10	78	210	406	666	990
Topology	*m:n*-2	*m:n*-3	*m:n*-4	*m:n*-5	*m:n*-6	*m:n*-7
Nodes	6	12	20	30	42	56
Edges	18	60	140	270	462	728

Table 2 Times in milliseconds required to solve the topologies

Topology	*1:n-s*-2	*1:n-s*-4	*1:n-s*-8	*1:n-s*-16	*1:n-s*-32	
Preparation	4.0 ± 0	5.6 ± 2	51 ± 2	950 ± 30	$24,000 \pm 200$	
Optimization	< 4.0	< 4.0	2.4 ± 2	33 ± 2	370 ± 5	
Topology	*1:n-r*-2	*1:n-r*-4	*1:n-r*-6	*1:n-r*-8	*1:n-r*-10	*1:n-r*-12
Preparation	4.0 ± 0	9 ± 2	39 ± 2	140 ± 6	410 ± 8	$1,300 \pm 20$
Optimization	< 4.0	5 ± 2	17 ± 3	120 ± 2	970 ± 10	$7,900 \pm 18$
Topology	*m:n*-2	*m:n*-3	*m:n*-4	*m:n*-5	*m:n*-6	*m:n*-7
Preparation	< 4.0	5.6 ± 2	16 ± 0	46 ± 3	120 ± 5	270 ± 2
Optimization	< 4.0	< 4.0	7.2 ± 2	18 ± 2	39 ± 3	100 ± 2

time is primarily caused by corresponding steep increase in the number of cycle-avoidance constraints (15). These were needed due to the increasing number of distributors. Similarly, the *m:n* topology can be solved for seven sites at most, as the *m:n*-8 topology requires many more distributors and consequently also cycle-avoidance constraints.

Compared to the previous CP-based solver [3], limitation in solving the *1:n-s* topologies stays roughly the same. However, most of the time is spent by the preparation phase, not by the IP solving. We will further pursue this issue to improve performance of the preparation phase for larger instances. The IP solver allows to solve the *1:n-r* topology for ten sites in real-time. This is a significant improvement compared to the CP-based solver, which allows for five sites at most. We also achieved an improvement for the *m:n* topologies, shifting from five to seven sites. The results might possibly be improved by a different formulation of cycle-avoidance.

5 Conclusions

Aiming to develop faster solver for the media streams planning problem, we presented the solution based on the integer programming model. Measured performance of the new solver shows promising results, shifting size of the problem instances which can be solved in real-time.

In our future work, we will evaluate performance of the solver more elaborately. We will also evaluate influence of the redundant constraints on performance of the backend solver. Further, we will explore another formulations of the cycle-avoidance constraints and the problem as a whole. Finally, we intend to explore various problem extensions to consider more general problems.

References

1 Ravindra K. Ahuja, Thomas L. Magnanti, and James Orlin. *Network Flows*. Prentice Hall, Englewood Cliffs, NJ, 1993.

2 Shiwen Chen, Oktay Günlük, and Bülent Yener. The multicast packing problem. *IEEE/ACM Transactions on Networking*, 8(3):311–318, 2000.

3 Petr Holub, Miloš Liška, and Hana Rudová. Data transfer planning with tree placement for collaborative environments, 2010. Under revision in *Constraints*.

4 Ayaz Isazadeh and Mohsen Heydarian. Optimal multicast multichannel routing in computer networks. *Computer Communications*, 31(17):4149 – 4161, 2008.

5 Miloš Liška and Petr Holub. CoUniverse: Framework for building self-organizing collaborative environments using extreme-bandwidth media applications. In *Euro-Par 2008 Workshops – Parallel Processing*, volume 5415 of *Lecture Notes in Computer Science*, pages 339–351. Springer, 2008.

6 Carlos A. S. Oliveira and Panos M. Pardalos. A survey of combinatorial optimization problems in multicast routing. *Computers & Operations Research*, 32(8):1953 – 1981, 2005.

7 Gábor Pataki. Teaching integer programming formulations using the traveling salesman problem. *SIAM Review*, 45:116–123, 2003.

8 Luca Sanna Randaccio and Luigi Atzori. Group multicast routing problem: A genetic algorithms based approach. *Computer Networks*, 51(14):3989–4004, 2007.

9 Helmut Simonis. Constraint applications in networks. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, pages 875–903. Elsevier, 2006.

10 Chu-Fu Wang, Chun-Teng Liang, and Rong-Hong Jan. Heuristic algorithms for packing of multiple-group multicasting. *Computers & Operations Research*, 29(7):905–924, 2002.

Monitoring and Control of Temperature in Networks-on-Chip

Tim Wegner, Claas Cornelius, Andreas Tockhorn, and Dirk Timmermann

Institute of Applied Microelectronics and Computer Engineering, University of Rostock
Richard-Wagner-Str. 31, 18119 Rostock-Warnemuende, Germany
`tim.wegner@uni-rostock.de`

Abstract

Increasing integration densities and the emergence of nanotechnology cause issues related to reliability and power consumption to become dominant factors for the design of modern multi-core systems. Since the arising problems are enforced by high circuit temperatures, monitoring and control of on-chip temperature profiles need to be considered during design phase as well as during system operation. Hence, in this paper different approaches for the realization and integration of a monitoring system for temperature in multi-core systems based on Networks-on-Chip (NoCs) in combination with Dynamic Frequency Scaling (DFS) are investigated. Results show that both combinations using event-driven and time-driven forwarding more than double overall execution time and considerably reduce throughput of application data. Regarding performance of notification and reaction to temperature development event-driven forwarding clearly outperforms time-driven forwarding.

Keywords and phrases Network-on-Chip, Reliability, Monitoring, Temperature, Control

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.124

1 Introduction

Aggressive downscaling of device sizes and ever increasing integration densities result in a fast growing number of processing and storage components per chip. This development gives rise to quickly growing systems with exceedingly high complexity, as it is well reflected in Systems-on-Chip (SoCs) combining multiple Intellectual Property (IP) cores. Against this background, NoCs provide an enabling solution to fulfill the communication requirements of such very-large-scale integrated systems [1]. However, this development causes issues related to reliability and robustness to become critical aspects for chip design. On the course of miniaturization, the transistor count per die increases, causing a generally higher probability of system failures on the one hand. On the other hand, probability for an individual transistor to fail is also raised, since the decreasing structural size of Integrated Circuits (ICs) leads to higher susceptibility to environmental influences and deterioration. Several physical mechanisms contributing to the these effects are known to be abetted by high temperatures. This leads to on-chip temperature distribution having considerable influence on various parameters of ICs like failure rate, lifetime, performance and power consumption. The correlation between temperature and deterioration is established by the Arrhenius model, describing the velocity of chemical reactions depending on temperature [10]. Two important mechanisms redounding to deterioration are Time Dependent Dielectric Breakdown (TDDB) and Electromigration (EM). TDDB describes the formation of charge traps in the gate oxide of a transistor and ultimately leads to gate oxide breakdown [9] rendering the transistor inoperative [3]. EM is defined as the transport of material caused by

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 124–131

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

ion movement in conductors and leads to the formation of locations of material loss (voids) and material accumulation (hillocks) [2]. Thus, current paths might be interrupted or short circuits might arise between adjacent wires.
The presented facts reason that adequate mechanisms for monitoring and control of on-chip temperature distribution are vitally important in order to mitigate the named effects and to delay failures caused by deterioration. Hence, in this paper methods to track temperature profiles of on-chip components and to react to temperature changes are examined with respect to the impact on performance of NoC-based multi-core systems, area costs and performance of temperature monitoring and control. More in detail, investigations are carried out for event-driven as well as time-driven forwarding of packets for temperature monitoring. Packets are sent to a Central Control Unit (CCU), which replies by giving instructions for DFS if necessary to lower the activity of the concerned IP core and thus relax its temperature.

2 Related Work

There has been a lot of work towards monitoring methods for NoC-based SoCs and management strategies for on-chip temperature, but little on the combination of the two delivering conclusions on the strengths and weaknesses of certain approaches. The concept of an event-based online monitoring service for NoCs is proposed in [4]. This service is based on reconfigurable event-based hardware probes attached to the NoC components (i.e. the routers) and offers runtime observability of the NoC behavior. In [5] the concept depicted above is examined on system level. Three different alternatives to integrate the monitoring service into a NoC are proposed and evaluated with respect to selected aspects. In [6] a NoC design flow, which takes monitoring into account at design time, is proposed. The paper focuses on the integration of monitoring into the overall NoC design process and makes proposals for the placement and the number of probes based on the underlying application. Moreover, design flow and system architecture for hierarchical power monitoring of on-chip networks are outlined in [7]. Thereto, a hierarchy of adaptive and scalable modules is used to handle various power monitoring services with different granularities. The hierarchy consists of multiple cell and cluster agents as well as a platform and an application agent. Similarly, in [8] a hierarchical agent-based concept is used to provide for reconfigurable NoCs with an increased fault tolerance on the architectural level. In case of a failure, communication and application execution are dynamically relocated based on given latency requirements.
In this paper the idea of a modular monitoring concept is adopted in order to monitor and control temperature in NoC-based SoCs. For this purpose, hardware probes, attached to all IP cores, forward the temperature to a CCU, which takes appropriate actions if necessary. Furthermore, values for area costs, system performance and performance of temperature monitoring and control are provided for both event-driven and time-driven temperature forwarding.

3 The Monitoring and Control System

In this section the monitoring system consisting of probes attached to every IP core and the CCU are introduced. The probes track the temperature of the associated IP core and the CCU gives instructions for DFS to the probes if necessary. Here, it is assumed that the temperature values are already available to the probes. Hence, this paper does not focus on the physical capture of the temperature value but on its further processing. The targeted NoC uses wormhole switching in combination with distributed XY-routing and

deploys a packet-based communication protocol, in which packets consist of a varying number of flow control digits (flits) representing the basic unit of flow control in the NoC. The target SoC is expected to be a general purpose system. This renders further assumptions on the incorporated applications dispensable. First, an event-driven and a time-driven solution for forwarding temperature values to the CCU are described. The targeted temperature range is from 20 to 127 °C. Furthermore, to ensure timely monitoring and control probe- and CCU-generated packets are prioritized over regular packets.

The event-driven probe design is based on [4]. The basic concept is that if the predefined conditions of an event are satisfied, further actions are triggered. The conditions here account for a maximum temperature or a large change within a given interval. The subsequent action is the forwarding of the received temperature value. In detail, the flow works as follows. The probe periodically reads temperature values and compares them to the value lastly reported to the CCU (T_{old}). If temperature forwarding is triggered, the temperature value T_{curr} is saved and a packet is generated containing T_{curr} as well as the address of the related IP core. This packet is then forwarded to the CCU. Figure 1 (a) depicts this flow exemplarily for a variation limit of $\Delta T \geq 10\,°C$, which can be freely chosen at design time in order to adapt the probe to different requirements and application settings. Relaxed conditions for ΔT will result in fewer monitoring packets less affecting regular traffic but leading to more intermittent temperature monitoring. If small changes of temperature shall be detected though, the conditions need to be more stringent leading to more packets potentially interfering with regular traffic. In general, a trade-off between quality and quantity (granularity of temperature forwarding versus number of generated packets) of event-driven temperature monitoring has to be made. Since packet generation for this approach depends on temperature gradients and therefore is non-deterministic, making statements about the optimal value for ΔT is nearly impossible. In contrast, the time-driven probe forwards an incoming temperature value periodically to the CCU based on a given period of time Δt. This is done independently from the current temperature, thus potentially causing redundant packets. The packet to be forwarded is identical to that from the event-driven approach. Figure 1 (b) illustrates the time-driven scheme of forwarding, in which Δt can be chosen at design time. This renders the time-driven approach deterministic as the amount of probe-generated traffic can be predicted in advance independently from temperature gradients. Therefore, Δt might be adopted to the prospective requirements.

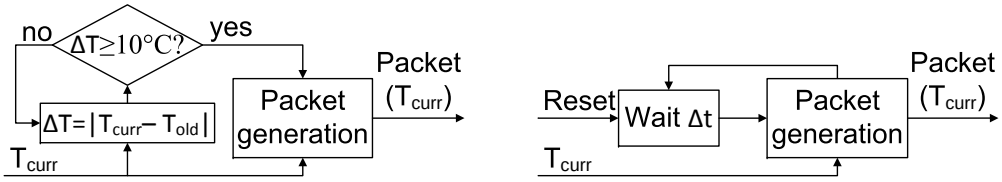


Figure 1 (a) Event-driven and (b) Time-driven temperature forwarding

All probe-generated packets are sent to the CCU, which is responsible for providing control instructions to the probes based on the reported temperatures. The frequency adjustment for an IP core is executed by the probe associated to this component once the probe has received instructions. Due to the fact that a considerable part of power consumed by ICs is dissipated as thermal energy, control instructions given by the CCU focus on methods to primarily reduce dynamic power consumption P_{dyn}. Based on the well-known equation $P_{dyn} = \alpha * \ C_L * \ V_{DD}^2 * \ f$, Dynamic Voltage and Frequency Scaling (DVFS) are the most

commonly used approaches. However, for the sake of simplicity, DVS is not implemented. Basically, the CCU waits until a packet arrives containing the temperature T_n of an IP core n. The desired operating frequency $f_{n,new}$ is then calculated based on the received temperature T_n and the current frequency $f_{n,curr}$. In case $f_{n,new}$ and $f_{n,curr}$ are identical, no action is taken. Otherwise, a packet containing DFS instructions is sent to the probe observing IP core n in order to reduce its activity and thus its temperature. Currently, the control mechanism is a simple reaction to temperature changes (DFS with five levels of frequency) to test the functionality of the probes. More advanced algorithms might be integrated prospectively considering the sophisticated correlations of temperature, reliability and further design metrics. The impacts on execution time of applications and operation conditions are not considered by the current simple DFS algorithm, which implies frequency reduction in case of raised temperatures as well as an incremental return to the maximum operating frequency in case of temperature normalization.
Since the probe is designed as an independent module, it has to be integrated accordingly into the NoC. Basically, three noteworthy possibilities exist. The probe's integration into an IP core promises to be the most straightforward and inexpensive solution as the probe uses the existing interface of the core for communication purposes (see Fig. 2 (a)). However, this precludes simultaneous packet transmission of probe and IP core. Furthermore, in case the IP core is unavailable (e.g. power down mode, failure), the probe becomes inaccessible and is no longer able to operate as it lacks of communication resources. Moreover, the integration of a probe into an IP core conflicts with the principle of strict modularity, which is one of the main intentions of NoCs. For the second approach the probe is placed outside the associated IP core (see Fig. 2 (b)). Here, both the probe and the IP core possess a dedicated communication interface, thus eliminating the effect of a probe being inoperable in case of core unavailability. Only the port of the router connecting the IP core to the NoC is shared among the probe and the core by using a Mux/Demux module. This module is responsible for correctly forwarding the traffic from and to the port of the router and always prioritizes probe-generated packets (current transmission of a packet containing multiple flits is finished first). Unfortunately, the Mux/Demux module and the additional interface induce extra area. As still only one router port is used, parallel communication of the IP core and the probe is not supported. The third alternative adds an extra port to the router in order to connect a probe to the NoC (see Fig. 2 (c)). Thereby, the probe is connected to the NoC completely independent from the IP core allowing full parallel communication, sustainment of modularity and operational readiness of the probe in case of IP core outage. The extra port and the raised complexity are supposed to induce the biggest area overhead though.

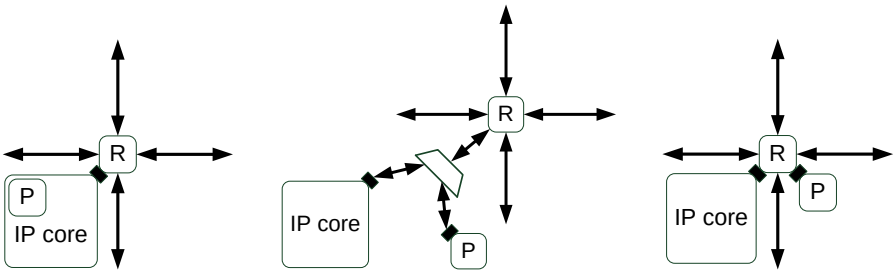


Figure 2 Integration of a probe (P) into the NoC: (a) Integration into the IP core, (b) Using the router port of the IP core, (c) Using an extra router port

4 Results and Discussion

All introduced proposals for probe design, the CCU and the integration into the NoC were synthesized with Xilinx Ise 10.1 for a Virtex5-FPGA to be comparable regarding area costs and frequency (see Tab. 1). Values for area and frequency of an unmodified NoC router serve as reference. Note that the required area is calculated as the number of pairs consisting of Look-Up Tables (LUT) and Flip-Flops (FF).

On the one hand, synthesis results for the temperature forwarding show that event-driven forwarding (66 LUT/FF pairs) occupies 17,5 % less area than the time-driven method (80 LUT/FF pairs). This is due to the fact that although more combinational logic is required by the event-driven probe, total area usage of the time-driven approach is dominated by the size of the counter that is necessary to trigger the periodic packet generation. To provide for a reasonable period, this results in a relatively large counter, which represents an area penalty compared to the event-driven scheme (e.g. $\Delta t = 2{,}5\,\text{ms}$ requires 18 bit wide counter based on a frequency of 100 MHz). Note that also existing trigger signals might be used for periodic packet generation. However, in this paper it is assumed that no such signals are available. On the other hand, reduced effort for combinational logic yields an operating frequency for the time-driven probe (338 MHz) that is 48,9 % faster with respect to the event-driven probe (227 MHz). From a cost-oriented point it can be concluded, that in case maximum frequency and determinism of packet-generation are the major design criteria, time-driven forwarding is the better choice. In case that minimal area and traffic overhead are required, event-driven forwarding is favored. Synthesis results for the CCU reveal that this module (507 LUT/FF pairs) requires only 28,6 % of the area needed for an router. Since an IP core is expected to require a multiple of this area, feasibility of positioning the CCU in a location regularly reserved for an IP core is affirmed. Admittedly, the CCU could also be implemented in software so that the kind of control could be adapted at runtime. However, the CCU is required to be implemented in hardware since the performance of communication between probes and CCU and its impact on system performance are of major interest in this paper. Due to its nature, the integration of a probe into an IP core causes no area overhead, whereas the maximum frequency of the network remains constant. Furthermore, the integration using the IP core's router port causes an area overhead of 7,34 % with respect to an unmodified router. Although the maximum frequency remains unchanged, the Mux/Demux module causes additional latency for packet transmission. The extra router port and the resulting raised complexity of the router not only cause the biggest area growth of 30,55 % but also lower the frequency to 112 MHz. Unfortunately, the most inexpensive method with the probe being integrated into the IP core cannot always be considered feasible, because IP cores often remain black boxes to the system designer when they are IP of third parties or fixed down to the layout. Regarding the two remaining possibilities, both methods do not necessitate IP core modification and guarantee probe operability in case the associated IP core becomes unavailable. However, the integration via an extra port causes unacceptable area overhead without delivering advantages (except parallel communication of probe and IP core) compared to the integration using the IP core's router port. Since the number of packets sent from and to the probe is comparatively small, the need for parallel communication is expected to be marginal. Therefore, the integration of a probe using the IP core's router port can be identified as the method delivering the best compromise between performance, overhead and feasibility.

Table 1 Synthesis results for monitoring, control as well as integration (unmodified NoC router: 1771 LUT/FF pairs, 122 MHz)

	Component			Integration method		
	Event-driven probe	Time-driven probe	Central Control Unit	Into IP core	Using IP core port	Extra port
Frequency [MHz]	227	338	165	122	119	112
Area [LUT/FF pairs]	66	80	507	1901	1896	2312

In order to investigate the impact of the examined methods on selected parameters of regular system operation, an 8×8 NoC was simulated both with and without enhancements for monitoring and control. Simulation was performed using both event-driven forwarding (with $\Delta T = 5\,°C$ and $10\,°C$) and time-driven forwarding (with $\Delta t = 0{,}1$ ms and 0,2 ms). The monitoring and control mechanisms were integrated using the approach proposed in Fig. 2 (b). During simulation 200000 regular packets with a maximum packet length of 5 flits were generated and sent to random destination addresses. Packet generation was uniformly distributed over all IP cores with an initial packet injection rate of 20 %. For simulation including monitoring and control the CCU replaced the most centric IP core at address 3,3 in the NoC. From Fig. 3 (a) it can be seen that mean packet delay of regular packets traversing the NoC is considerably decreased both for event-driven (up to 9,72 %) and time-driven (8,53 %) temperature forwarding combined with DFS. At first sight, this seems to be incorrect, since packets for monitoring and control additionally stress the NoC and therefore should have negative impact. Considering the overall execution time, which is drastically extended by at least 125,16 % for all tests, this phenomenon can be explained. In contrast to the reference, the applied DFS mechanism reduces the injection rate of the IP cores if necessary. Therefore, the overall number of packets (monitoring and control packets included) simultaneously crossing the NoC is reduced, yielding relaxed conditions for packet transmission. Thus, a lower utilization of the communication resources is traded off against an extension of overall execution time. This conforms to the principle of algorithms like DFS of maintaining operability at the expense of reduced performance. As expected, with a reduction of more than 55 %, integration of monitoring and control has a negative impact on the throughput of regular data, since fewer packets located in the NoC lead to a smaller number of flits that can be transmitted per clock cycle. Concluding, concerning effects on performance event-driven and time-driven forwarding do not differ notably from each other when combined with DFS. Furthermore, variation of ΔT (event-driven) and Δt (time-driven) has only minor influence on performance parameters. Concerning the results for notification delay (time for packet transmission from probes to CCU) and response delay (notification delay + time for packet transmission from CCU to probes) the event-driven approach outperforms the time-driven approach (see Fig. 3 (b)). In the former packets for notification arrive about 83 % faster than in the latter and response packets arrive about 77 % faster. This is due to the fact that for event-driven forwarding notification packets are only generated when temperature exceeds or goes below defined thresholds. This leads to a relatively even distribution of monitoring packets over time avoiding congestions around the CCU. In contrast, in the time-driven approach all probes simultaneously transmit a packet to the CCU after Δt has expired. As a consequence, bursty occurrence of monitoring

packets causes congestions within the area containing the CCU. To solve this issue the probes' counters for Δt might be reset at different points in time resulting in timely staggered generation of monitoring packets. Although delays for time-driven triggering might be significantly reduced hereby, the risk of transmitting identical temperature values repeatedly and causing unnecessary traffic remains. Therefore, the event-driven approach promises to be the more practicable method.

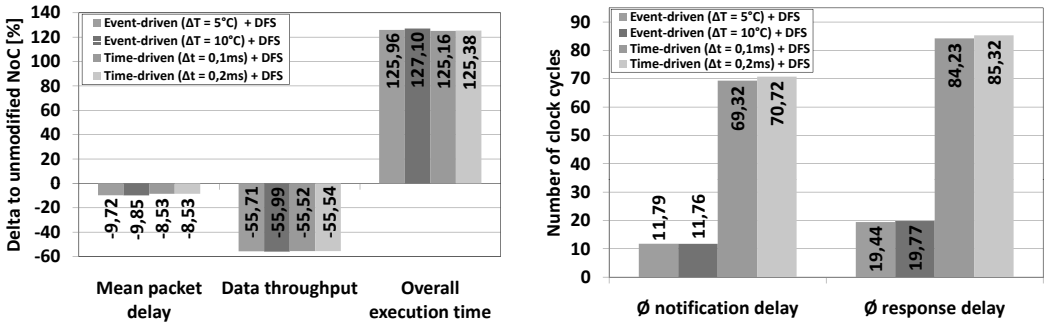


Figure 3 (a) Performance results for a NoC enhanced by monitoring and control in comparison with an unmodified NoC, (b) Delay results for notification (packets from probes to CCU) and response (packets from probes to CCU and back to probes)

5 Conclusion

In this paper the idea of a modular monitoring concept for NoCs is adopted for temperature monitoring in NoC-based SoCs and combined with DFS for control. For this purpose, probes inherit the function of monitoring temperature of the System-on-Chip, consisting of IP cores, and a CCU assumes the task of applying DFS to the IP cores. Regarding the integration of monitoring and control into the NoC we argued that using an IP core's router port poses the best trade-off between feasibility, performance and additional costs. Furthermore, an event-driven and a time-driven approach for forwarding temperature values from probes to CCU in combination with DFS were examined regarding their impact on performance of regular system operation and monitoring and control. Results show that both approaches similarly extend overall execution time by up to 127,1 % and reduce throughput of application data by up to 55,99 %. In return NoC utilization is reduced and mean packet delay is decreased by up to 9,85 %. Using the event-driven approach packet delay from probes to CCU is shortened by almost 83 % and response delay (packet from probe to CCU back to probe) is abbreviated by 77 % with respect to time-driven forwarding.

References

1 L. Benini and G. De Micheli, "Networks on Chips: A New SoC Paradigm", IEEE Computers, Jan. 2002, pp. 70-78.
2 J. Lienig, "Introduction to Electromigration-Aware Physical Design", in Proc. of ISPD 2006.
3 E. Vogel, et al., "Reliability of Ultra-Thin Silicon Dioxide Under Combined Substrate Hot Electron and Constant Voltage Tunnel Stress", in Trans. of Electron Devices, vol. 47, no. 6, 2000.
4 C. Ciordas, et al., "An Event-based Monitoring Service for Networks on Chip", in ACM TOADES 2005, vol. 10, no. 4, pp. 702-723.

5 C. Ciordas, et al., "NoC Monitoring: Impact on the Design Flow", in Proc. of IEEE ISCAS 2006.

6 C. Ciordas, et al., "A Monitoring-Aware Network-on-Chip Design Flow", J. Syst. Archit., vol. 54, issue 3-4 (March 2008), pp. 397-410.

7 L. Guang, et al., "Hierarchical Power Monitoring for On-chip Networks", in Proc. of PDP 2009.

8 P. Rantala, et al., "Novel Agent-Based Management for Fault-Tolerance in Network-on-Chip", in Proc. of DSD 2007.

9 J. Stathis, et al., "Reliability Limits for the Gate Insulator in CMOS Technology", IBM J. of Research and Development, 2002.

10 J. Srinivasan, et al., "RAMP: A Model for Reliability Aware Microprocessor Design", IBM Research Report, RC23048, 2003.