Model-Based Hardware Design based on Compatible Sets of Isomorphic Subgraphs

Patrick Sittel, Konrad Möller, Martin Kumm, Peter Zipf
University of Kassel
{sittel,konrad.moeller,kumm,zipf}@uni-kassel.de

Bogdan Pasca, Mark Jervis
Intel Corperation
{bogdan.pasca, mark.jervis}@intel.com

***Abstract*—Hardware applications in an industrial context often have tight area, latency and throughput requirements or a specific combination thereof. This paper presents a method to improve area and throughput figures for folded circuits generated during a model-based hardware design process. The method targets FPGA implementations and is based on the automatic combination of isomorphic subgraphs and the detailed consideration of pipelined primitive operations for folding core scheduling. In the course of a design space exploration, the user is provided with fine-grain control over the area/throughput trade-off.**

I. Introduction

Model-based design (MBD) tools, like the Matlab/Simulink HDL Coder or the Intel DSPBuilder for FPGAs, are a de-facto standard in digital signal processing (DSP), control theory and embedded system development. Exploiting a high level of abstraction, MBD tools aid system designers during the modelling, implementation and testing process, thus enabling more complex designs and reducing the time-to-market [14]. Compact hardware implementations can be generated using time-multiplexed resource sharing, often called folding [10].

Casseau and Le Gal [4] described a model-based high-level synthesis (HLS) flow for designing FPGA-based DSP systems using a combined scheduling and binding algorithm for single operations. But, with an identification and selection by hand, it could be shown that it is beneficial for hardware reduction and throughput improvement to consider the use of isomorphic and connected subgraphs for folding [8]. The automatic identification of isomorphic subgraphs in the context of model-based design was shown in a subsequent work [13]. However, a combination and selection algorithm for folding using compatible isomorphic subgraphs was not proposed. Combining operations that occur in identical patterns to form larger functional units for resource sharing has also been discussed in the field of designing application-specific instruction-set processors (ASIPs) [9]. Pozzi et al. [12] described exact and approximate speed-up based subgraph selection algorithms for instruction set extension. As our primary goal is area reduction, we propose to select the combination of subgraph groups with the best estimated area savings, avoiding multiple selection iterations. Clark et al. [5] and Bonzini et al. [1] proposed tool flows based on enumerating all subgraphs, before graph isomorphisms are detected. In this paper, we use an algorithm that enumerates frequent subgraphs exclusively, which is favorable in terms of algorithm runtime.

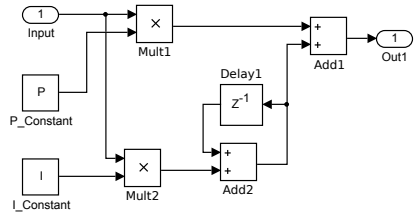


Fig. 1: Simulink description of a PI controler

Cong and Jiang proposed an heuristic approach to recognize isomorphic subgraphs [6]. The authors define the occurrences of a pattern as a pattern instance and a set of pattern instances as a pattern group. Still, the problem of combining pattern instances and pattern groups remains a not completely solved problem, because conflicting combinations of pattern groups are not further addressed.

Our previous work [13] and an analysis of Intel DSPBuilder results show that many desirable solutions in the design space are not reached, because of inefficient scheduling and the lack of pipelining. To the best of our knowledge, efficient scheduling using pipelined folding cores, consisting of isomorphic subgraphs, was not considered in previous work. In this paper, we propose an algorithm for combining isomorphic subgraphs and groups of isomorphic subgraphs for folding and a scheduling method that considers resource sharing with isomorphic subgraphs that consist of pipelined operations.

II. Motivational Example

Figure 1 shows the Simulink model of a proportional-integral (PI) controler. The initial folding approach of Parhi [10] uses the single operations of $\{Add\}$ and $\{Mult\}$ as folding cores. The folded circuit using these folding cores, highlighted in blue, is shown in Figure 2. The number of adders and multipliers used could each be reduced by one, but additional multiplexers, registers and a control counter, highlighted in red, have to be implemented.

A model that was generated by using isomorphic subgraphs for folding is shown in Figure 3. The two embeddings of the subgraph $\{Mult, Add\}$ were used for resource sharing. This allows reducing the multiplexer and register counts by one, thus resulting in a more efficient implementation. The direct connection between the adder and the multiplier has two benefits. On the one hand it saves the adder input multiplexer,

978-1-5090-4825-0/17/$31.00 ©2017 IEEE

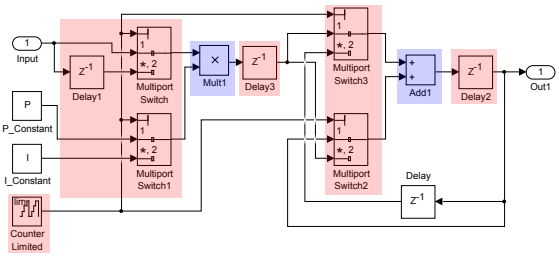


Fig. 2: PI controler folded with single operations

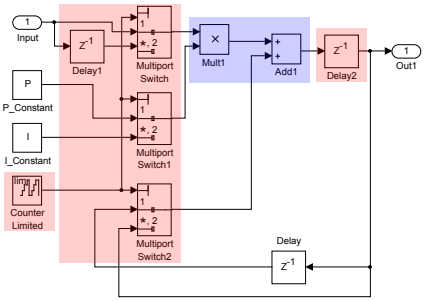


Fig. 3: PI controler folded with the subgraph (Mult,Add)

thus saving resources. On the other hand, it also allows removing the *Delay3* register which saves resources, but also shortens the overall schedule length. The savings obtained on this example motivate our work in identifying and using larger isomorphic subgraphs for folding.

III. Problem Formulation

Analog to Bringmann and Nijssen [2], we use compatibility and overlap graphs to formulate the problem of isomorphic subgraph combination for folding.

Definition 1: Let $G = (V_G, E_G, l_G)$ denote the DFG representation of the input model, consisting of a set of vertices V_G, directed edges E_G and a labeling function for vertices l_G, denoting operator types. Every connected $p = (V_p, E_p, l_p)$ is called a pattern occurring in G if there exists a function $\rho : V_p \rightarrow V_G$ mapping the vertices of p to the according vertices of G such that $(I) \forall\, v \in V_p \Rightarrow l_p(v) = l_G(\rho(v))$ and $(II) \forall\, (u,v) \in E_p \Rightarrow (\rho(v), \rho(u)) \in E_G$. Then every subgraph $\delta^p = (V_{\delta^p} \subseteq V_G, E_{\delta^p} \subseteq E_G, l_G)$ in G such that $(I) \forall\, v \in V_{\delta^p} \Rightarrow l_p(v) = l_G(\rho(v))$ and $(II) \forall\, (u,v) \in E_p \Rightarrow (\rho(v), \rho(u)) \in E_{\delta^p}$ is defined as an **occurrence** of p in G.

An occurrence is an embedding of a pattern into G. Multiple occurrences of a pattern are used for resource sharing and folding core generation is strongly related to the problem of frequent subgraph or pattern mining [15]. A pattern is considered to be frequent in G if it has multiple, i.e., a user-defined integer threshold, occurences in G. Applying a frequent subgraph mining algorithm, the set of all frequent patterns $\mathcal{P} = \{p_i \mid i = 1, ..., n\}$ and the set of all occurrences $\Delta = \{\Delta^{p_i} \mid p_i \in \mathcal{P}\}$, where Δ^{p_i} contains all occurrences of p_i in G, are identified.

Definition 2: Two occurrences δ_j, δ_k of arbitrary patterns are defined as **compatible** or **conflict free** if and only if the intersection of their node set is empty, i.e. $V_{\delta_j} \bigcap V_{\delta_k} = \emptyset$.

Note that δ_j and δ_k may be be isomorphic. In the following, we use an undirected graph to describe the compatibility of occurrences in Δ^{p_i}. A compatibility graph of Δ^{p_i} is a graph $\mathrm{CG}^{p_i}_{\mathrm{occ}} = (V_{p_i}, E_{p_i})$ where V_{p_i} is the set of all occurrences δ^{p_i} of p_i in G and $(\delta^{p_i}_j, \delta^{p_i}_k) \in E_{p_i}$ if and only if they are compatible.

Definition 3: The **minimum support** $S_{\min}$ (or minimum frequency) of a pattern p in G is defined as the maximum number of compatible occurrences of p in G.

Bringmann and Nijssen proved that $S_{\min}$ is the size of a maximal clique in $\mathrm{CG}_{\mathrm{occ}}$. In the following, we will use cliques in $\mathrm{CG}_{\mathrm{occ}}$ to combine isomorphic occurrences of patterns for resource sharing by applying the folding transformation.

Definition 4: Given a pattern p in G and a set of all occurrences Δ^p of p in G. Then, we denote each subset $\Delta^p_k \subseteq \Delta^p$, with $\mid \Delta^p_k \mid \geq 2$, of pairwise compatible occurrences as a **folding core**.

Occurrences that share operations in the input graph can not be used in combination to generate a folding core. Therefore, Definition 4 is an important addition to the formulation of Parhi, because single operations are never incompatible to each other and always form a valid folding core. Definition 4 ensures the generation of valid folding cores when using isomorphic subgraphs.

Definition 5: The size $N^p_k = \mid \Delta^p_k \mid$ of the k-th folding core Δ^p_k of a pattern p is called the **folding factor** of Δ^p_k.

The folding factor describes the serialization of the DFG after resource-sharing is applied. A larger folding factor means that more occurrences of a subgraph in the input model are used for time-multiplexed resource-sharing.

Definition 6: Given any two patterns p_1, p_2 in G and two sets of all occurrences Δ^{p_1}, Δ^{p_2}. Then, two folding cores $\Delta^{p_1}_k$ and $\Delta^{p_2}_l$ are defined as **compatible** if and only if all $\delta^{p_1}_m \in \Delta^{p_1}_k$ and $\delta^{p_2}_n \in \Delta^{p_2}_l$, with $0 \leq m < N^{p_1}_k$ and $0 \leq n < N^{p_2}_l$ are pairwise compatible.

Given a set of folding cores $\mathcal{F} = \{\Delta^{p_i}_k \mid p_i \in \mathcal{P}, \Delta^{p_i}_k \subseteq \Delta^{p_i}\}$, a compatibility graph of folding cores is a graph $\mathrm{CG}_{\mathrm{fc}} = (V_{\mathcal{F}}, E_{\mathcal{F}})$ where $V_{\mathcal{F}}$ is set of all $\Delta^{p_i}_k \in \mathcal{F}$ and $(\Delta^{p_i}_k, \Delta^{p_j}_l) \in E_{\mathcal{F}}$ if and only if they are compatible. The vertices of $\mathrm{CG}_{\mathrm{fc}}$ each represent a clique in the constituting compatibility graphs $\mathrm{CG}^{p_i}_{\mathrm{occ}}$ of each pattern $p_i \in \mathcal{P}$.

Definition 7: A set ϕ of pairwise compatible folding cores is denoted as a **folding core combination**.

A set of folding core combinations $\{\phi_d \mid d = 1, ..., n\}$ is denoted as Φ. Each ϕ_d is a clique in $\mathrm{CG}_{\mathrm{fc}}$ and describes one way for folding the input model using isomorphic subgraphs. Each folding core combination can be seen as a mapping of elementary blocks in the input model to an occurrence and folding core. The problem of folding using isomorphic subgraphs can now be formulated as follows:

Problem: Given an input model m, determine all Pareto optimal implementations of combinations $\phi_d \in \Phi$ of m, with respect to hardware effort and throughput.

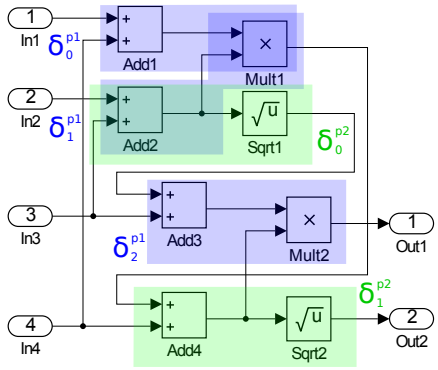
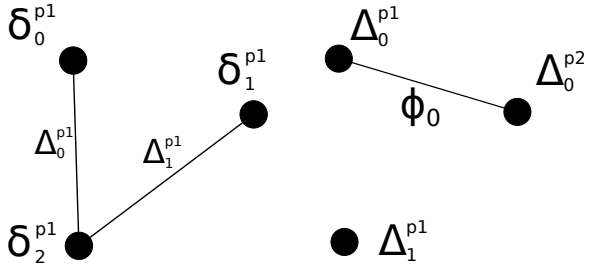


Fig. 4: Folding core combination example



(a) $CG_{occ}^{p_1}$: Compatible folding cores of the pattern p_1

(b) CG_{fc}: Compatible folding core combinations

Fig. 5: Folding core generation and combination

IV. Subgraph Combination

In general, many occurrences of patterns are enumerated and combining them is a central task of our approach. We divide the process of subgraph combination into two steps using compatibility graphs and cliques. First of all, subgraph based folding cores have to be generated. In the second step, these folding cores are used to generate folding core combinations. The following example illustrates this concept.

The Simulink model depicted in Figure 4 performs four additions, two multiplications and two square root computations. The pattern $p_1 = \{Add, Mult\}$ has three occurrences $\delta_0^{p_1} = \{Add1, Mult1\}$, $\delta_1^{p_1} = \{Add2,\ Mult1\}$ and $\delta_2^{p_1} = \{Add3, Mult2\}$. Using p_1, it is possible to generate two different folding cores: $\Delta_0^{p_1} = \{\delta_0^{p_1}, \delta_2^{p_1}\}$ and $\Delta_1^{p_1} = \{\delta_1^{p_1}, \delta_2^{p_1}\}$. According to Definition 4, it is not allowed to combine all three occurrences of p_1 to achieve a folding factor of three, because $\delta_0^{p_1}$ and $\delta_1^{p_1}$ are incompatible, as they share $Mult1$.

The folding cores $\Delta_0^{p_1}$ and $\Delta_1^{p_1}$ from Figure 4 are incompatible, because both of them contain $\delta_2^{p_1}$. But another pattern $p_2 = \{Add, Sqrt\}$ can be found in Figure 4 and be used for folding. The occurrences of p_2 are $\delta_0^{p_2} = \{Add2, Sqrt1\}$, $\delta_1^{p_2} = \{Add4, Sqrt2\}$ and it is possible to generate one folding core $\Delta_0^{p_2} = \{\delta_0^{p_2}, \delta_1^{p_2}\}$. Since $\delta_0^{p_2}$ is incompatible to $\delta_1^{p_1}$, $\Delta_0^{p_2}$ can not be used for folding in combination with $\Delta_1^{p_1}$, however it is compatible to $\Delta_0^{p_1}$ and a folding core combination $\phi_0 = \{\Delta_0^{p_1}, \Delta_0^{p_2}\}$ is found.

Figure 5 shows the compatibility graphs of the discussed folding cores and folding core sets for Figure 4. All occurrences of p_1 are visualized as a node in Figure (5a) and every edge in this graph describes a compatibility of occurrences. Each clique that contains at least two vertices in $CG_{occ}^{p_1}$ describes a folding core $\Delta_k^{p_1}$. The compatibility of different folding cores is shown in Figure (5b). In this graph, every clique describes a possible folding core combination.

Algorithm 1 Generation of Folding Core Combinations

Input: $\mathcal{P} = \{p_i \mid i = 1, ..., n\}$, $\Delta = \{\Delta^{p_i} \mid p_i \in \mathcal{P}\}$
Output: Φ_{MC}

1: **Declare** an empty set of folding cores $\mathcal{F}$
2: **for** `each` $p_i \in \mathcal{P}$ **do**
3: $\quad \Delta_0^{p_i}$ = oneMaximalClique(Δ^{p_i})
4: $\quad \mathcal{F} \leftarrow \mathcal{F} \cup \Delta_0^{p_i}$
5: **end for**
6: $CG_{\mathtt{fc}}$ = compatibilityGraph($\mathcal{F}$)
7: Φ_{MC} = allMaxCliques($CG_{\mathtt{fc}}$)
8: **return** Φ_{MC}

Algorithm 1 describes the generation of folding core combination as pseudo code. Given a set of frequent patterns and their occurrences, Algorithm 1 returns a number of valid folding core combinations. First, one maximal clique from each subgraph's occurrences is generated in line 3 and added to the set of folding cores $\mathcal{F}$ in line 4. Using $\mathcal{F}$, a compatibility graph CG_{fc} is generated in line 6. At last, all maximal cliques [3] of CG_{fc} are determined and returned in line 7.

The proposed combination algorithm is a heuristic approach. Only one maximal clique is used as folding core and all other cliques are pruned. Additionally, only maximal cliques (MC) are considered to generate folding core combinations Φ_{MC} in line 7. This means that all subcliques that generate more parallel hardware using a subset of folding cores are pruned.

V. Folding Core Scheduling

Figure 6 shows an example circuit where two occurrences of a pattern $p = \{Add, Mult\}$ are highlighted in red (δ_0^p) and green (δ_1^p) and two remaining elementary blocks $SQRT$ and $Gain$ are highlighted in black and blue. The number of internal pipeline stages are denoted in each operations' bottom right. A naive method that is shown in the top half of Figure 7 schedules all inputs of δ_0^p at clock cycle 4, because $Mult1$ can only be scheduled after $SQRT$ is computed. In total, the schedule length using the naive method is 19 time steps. Systematically, the schedule length can be reduced to 13, when the folding core input and output ports have the flexibility of being scheduled independently. The result is shown in the bottom half of Figure 7. Respecting the pipeline stages and connections, the input ports of δ_0^p are scheduled at different time steps. This concept is also applied for δ_1^p, resulting in an overall schedule that is 6 cycles shorter.

VI. Experimental Results

Synthesis experiments were done using the DSP Builder for Intel FPGAs for a description of the Park-Clark Transformation (PCT) [11] and an Arria10 FPGA (10AX066K2F35I2LG). The circuits using isomorphic subgrpahs and a naive schedule

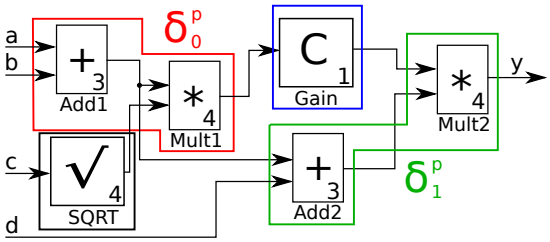


Fig. 6: Folding core scheduling example

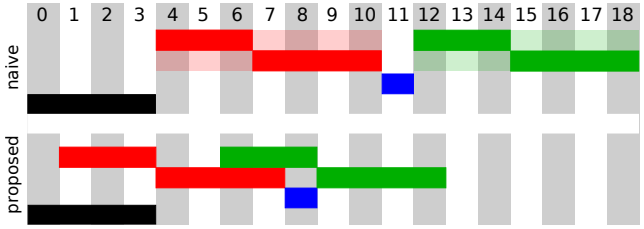


Fig. 7: Folding Core Scheduling

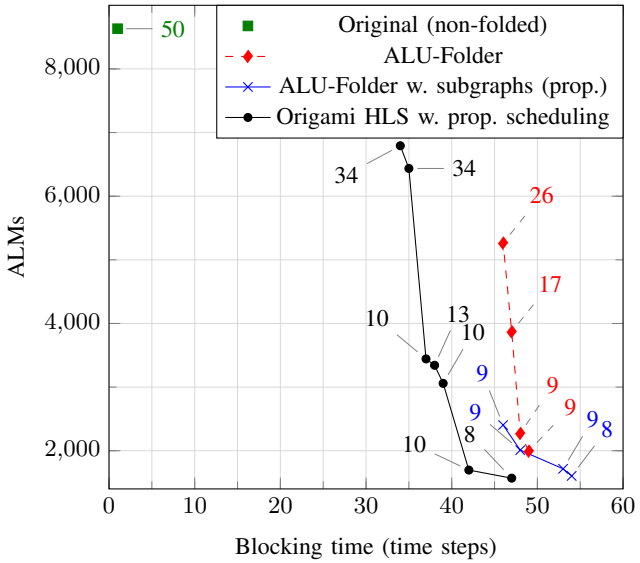


Fig. 8: Pareto front implementations of PCT

are implemented in the ALU-Folder. The solutions using the above described scheduling method were generated with our open-source tool Origami HLS [7]. For all experiments, the minimum frequency was set to 100 MHz.

Depending on the schedule, new data samples can only be inserted after a specific number of time steps. We call this period, the blocking time of the pipeline. Figure 8 shows the hardware costs in ALMs over the blocking time of the folded circuits as time steps. The occupied DSP blocks are assigned to each point. In the contrast to the ALU-Folder, Origami HLS is able to identify parts of the circuit that are not used for folding. Compared to the ALU-Folder, the next sample can be inserted earlier, resulting in a higher throughput. The non folded PCT model implementation is displayed by the upper left green point. The implementations that were generated using the actual version of the ALU-Folder are displayed as red rhombuses. The solutions displayed as blue crosses were generated using isomorphic subgraphs. The solutions displayed as black circles were generated using Origami HLS that uses the proposed scheduling. The blocking time of the non-folded model is one time step. The highest throughput for the current version of the ALU-Folder or the ALU-Folder with isomorphic subgraphs is achieved with a blocking time of 46 time steps. In all observed cases, the proposed folding core scheduling further reduces blocking time, thus providing better solutions with respect to area and throughput.

VII. Conclusion

We presented an ALU-Folder extension for MBD that generates folding core combinations based on isomorphic subgraphs with pipelined operations. Better FPGA implementations, which operate at reasonable frequencies ($>$ 100 MHz) and require up to 54.3% fewer ALMs and 65.4% DSP blocks compared to the current version of the ALU-Folder could be generated. Additionally, it could be shown that the presented scheduling approach reduces the blocking time.

References

[1] P. Bonzini and L. Pozzi. Recurrence-Aware Instruction Set Selection for Extensible Embedded Processors. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 16(10):1259–1267, 2008.

[2] B. Bringmann and S. Nijssen. What is Frequent in a Single Graph? In *Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, pages 858–863. Springer, 2008.

[3] C. Bron and J. Kerbosch. Algorithm 457: Finding All Cliques of an Undirected Graph. *Communications of the ACM*, 16(9):575–577, 1973.

[4] E. Casseau and B. Le Gal. High-Level Synthesis for the Design of FPGA-based Signal Processing Systems. In *Internat. Symp. on Systems, Architectures, Modeling, and Simulation*, pages 25–32. IEEE, 2009.

[5] N. Clark, H. Zhong, and S. Mahlke. Processor Acceleration through Automated Instruction Set Customization. In *Proc. of the 36th annual IEEE/ACM Internat. Symp. on Microarchitecture*, page 129. IEEE Computer Society, 2003.

[6] J. Cong and W. Jiang. Pattern-based Behavior Synthesis for FPGA Resource Reduction. In *16th Internat. Symp. on Field programmable Gate Arrays*, pages 107–116. ACM, 2008.

[7] M. Kumm, P. Sittel, K. Möller, M. Hardieck and P. Zipf. Origami HLS. http://www.uni-kassel.de/go/origami, 2016.

[8] K. Möller, M. Kumm, C.-F. Müller, and P. Zipf. Model-Based Hardware Design for FPGAs using Folding Transformations based on Subcircuits. *FPGAs for Software Programmers (FSP)*, 2015.

[9] N. Moreano, E. Borin, C. De Souza, and G. Araujo. Efficient Datapath Merging for Partially Reconfigurable Architectures. *IEEE Trans. on Comput.-Aided Design Integr. Circuits Syst.*, 24(7):969–980, 2005.

[10] K. K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley & Sons, 2007.

[11] R. H. Park. Two-Reaction Theory of Synchronous Machines Generalized Method of Analysis-Part I. *Trans. of the American Institute of Electrical Engineers*, 48(3):716–727, 1929.

[12] L. Pozzi, K. Atasu, and P. Ienne. Exact and Approximate Algorithms for the Extension of Embedded Processor Instruction Sets. *IEEE Trans. on Comput.-Aided Design Integr. Circuits Syst.*, 25(7):1209–1229, 2006.

[13] P. Sittel, M. Kumm, K. Möller, M. Hardieck, and P. Zipf. High-Level Synthesis for Model-Based Design with Automatic Folding including Combined Common Subcircuits. *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 103–114, 2017.

[14] W. Wu. *Model-Based Design for Effective Control System Development*. IGI Global, 2017.

[15] X. Yan and J. Han. gSpan: Graph-based Substructure Pattern Mining. In *Internat. Conf. on Data Mining*, pages 721–724. IEEE, 2002.