Proceedings of the 14th Annual International Converence on Field-Programmable Logic and Applications (FPL '04)

Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs

Zachary K. Baker and Viktor K. Prasanna

University of Southern California, Los Angeles, CA, USA
zbaker@halcyon.usc.edu, prasanna@ganges.usc.edu

Abstract. This paper presents a tool for automatic synthesis of highly efficient intrusion detection systems using a high-level, graph-based partitioning methodology, and tree-based lookahead architectures. Intrusion detection for network security is a compute-intensive application demanding high system performance. This tool automates the creation of efficient FPGA architectures using system-level optimizations, a relatively unexplored field in this area. The pre-design tool allows for more efficient communication and extensive reuse of hardware components for dramatic increases in area-time performance. The tool is available online for public use.

1 Introduction

Pattern matching for network security and intrusion detection demands exceptionally high performance. This performance is dependent on the ability to match against a large set of patterns, and thus the ability to automatically optimize and synthesize large designs is vital to a functional network security solution. Much work has been done in the field of string matching for network security [1–5]. However, the study of the *automatic design* of efficient, flexible, and powerful system architectures is still in its infancy.

Snort, the open-source IDS [6], and Hogwash [7] have thousands of content-based rules. A system based on these rulesets requires a hardware design optimized for thousands of rules, many of which require string matching against the entire data segment of a packet. To support heavy network loads, high performance algorithms are required to prevent the IDS from becoming the network bottleneck. One option is to move the matching away from the processor and on to an FPGA, wherein a designer can take advantage of the reconfigurability of the device to produce customized architectures for each set of rules.

By carefully and intelligently processing an entire ruleset (Figure 1), our tool can *partition* a ruleset into multiple pipelines in order to optimize the area and time characteristics of the system. By applying automated graph theory and trie techniques to the problem, the tool effectively optimizes large ruleset, and then generates a fully synthesizeable architecture description in VHDL ready for

[1] Supported by the United States National Science Foundation/ITR under award No. ACI-0325409 and in part by an equipment grant from the HP Corporation.

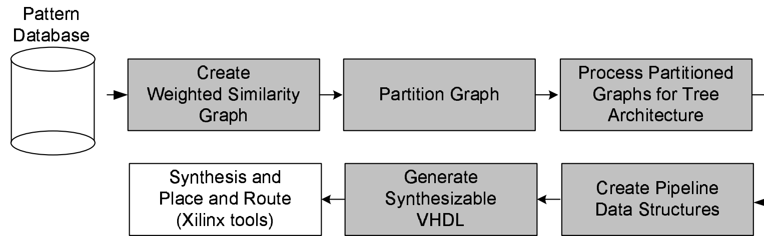

Fig. 1. Automated optimization and synthesis of partitioned system

place and route and deployment in less than 10 seconds for a set of 351 patterns, and less than 30 for 1000 patterns.

2 Related Work in Automated IDS Generation

Snort [6] and Hogwash [7] are current popular options for implementing intrusion detection in software. They are open-source, free tools that promiscuously tap the network and observe all packets. After TCP stream reassembly, the packets are sorted according to various characteristics and, if necessary, are string-matched against rule patterns. However, the rules are searched in software on a general-purpose microprocessor. This means that the IDS is easily overwhelmed by periods of high packet rates. The only option given by the developers to improve performance is to remove rules from the database or allow certain classes of packets to pass through without checking. Some hacker tools even take advantage of this weakness of Snort and attack the IDS itself by sending worst-case packets to the network, causing the IDS to work as slowly as possible. If the IDS allows packets to pass uninspected during overflow, an opportunity for the hacker is created. Clearly, this is not an effective solution for maintaining a robust IDS.

Automated IDS designs have been explored in [1], using automated generation of Non-deterministic Finite Automata. The tool accepts rule strings and then creates pipelined distribution networks to individual state machines by converting template-generated Java to netlists using JHDL. This approach is powerful but performance is reduced by the amount of routing required and the logic complexity required to implement finite automata state machines. The generator can attempt to reduce logic burden by combining common prefixes to form matching trees. This is part of the pre-processing approach we take in this paper.

Another automated hardware approach, in [5], uses more sophisticated algorithmic techniques to develop multi-gigabyte pattern matching tools with full TCP/IP network support. The system demultiplexes a TCP/IP stream into several substreams and spreads the load over several parallel matching units using Deterministic Finite Automata pattern matchers. In their architecture, a web interface allows new patterns to be added, and then the new design is generated and a full place-and-route and reconfiguration is executed, requiring 7-8 minutes. As their tools have been commercialized in [8], only some of their architectural components are freely available to the community.

The NFA concept is updated with predecoded inputs in [9]. This paper addresses the poor frequency performance as the number of patterns increases, a weakness of earlier work. This paper solves most of these problems by adding predecoded wide parallel inputs to a standard NFA implementations. The result is excellent area and throughput performance.

In [2, 3], a hardwired design is developed that provides high area efficiency and high time performance by using replicated hardwired 32-bit comparators in a pipeline structure. The matching technique proposed is to use four 32-bit hardwired comparators, each with the same pattern offset successively by 8 bits, allowing the running time to be reduced by 4x for an equivalent increase in hardware. These designs have adopted some strategies for reducing redundancy through pre-design optimization. Expanding their earlier work, [2] reduces area by finding identical alignments between otherwise unattached patterns. Their preprocessing takes advantage of the shared alignments created when pattern instances are shifted by 1, 2, and 3 bytes to allow for the 32-bit per cycle architecture. The work in [3] shares the pre-decoded shift-and-compare approach with our work, but they utilize SRL16 shift registers where we utilize single-cycle delay flip-flops. Their work also utilizes a partitioning strategy based on incrementally adding elements to partitions to minimize the addition of new characters to a given partition.

3 Our Approach

This research focuses on the automatic optimization and generation of high-performance string-matching systems that can support network traffic rates while providing support for large pattern databases. The tool generates two basic architectures, a pre-decoded shift-and-compare architecture, and a variation using a tree-based area optimization. In this architecture, a character enters the system and is "pre-decoded" into its character equivalent. This simply means that the incoming character is presented to a large array of AND gates with appropriately inverted inputs such that the gate output asserts for a particular character. The outputs of the AND gates are routed through a shift-register structure to provide time delays. The pattern matchers are implemented as another array of AND gates and the appropriate decoded character is selected from each time-delayed shift-register stage. The tree variation is implemented as a group of inputs that are pre-matched in a "prefix lookahead" structure and then fed to the final matcher stage. The main challenge in the tree structure is creating the trees; this is discussed in Section 4.

The notion of predecoding has been explored in [10] in the context of finite automata, the use of large, pipeline brute-force comparators for high speed was initiated by [11] and furthered by [12]. The use of trees for building efficient regular expression state machines was initially developed by [1]. We explored the partitioning of patterns in the pre-decoded domain in [13]. We utilize these foundational works and build automatic optimization tools on top.

With our domain specific analysis and generation tool, extensive automation carefully partitions a rule databases into multiple independent pipelines. This allows a system to more effectively utilize its hardware resources. The key to our

performance gains is the idea that characters shared across patterns do not need to be redundantly compared. Redundancy is an important idea throughout string matching; the Knuth-Morris-Pratt algorithm, for instance, uses precomputed redundancy information to prevent needlessly repeating comparisons. We utilize a more dramatic approach; by pushing all character-level comparisons to the beginning of the comparator pipelines , we reduce the character match operation to the inspection of a single bit.

Increasing the number of bits processed at a single comparator unit increases the delay of those gates. The pre-decoding approach moves in the opposite direction, to single-bit, or *unary*, comparisons. We decode an incoming character into a "one-hot" bit vector, in which a character maps to a single bit.

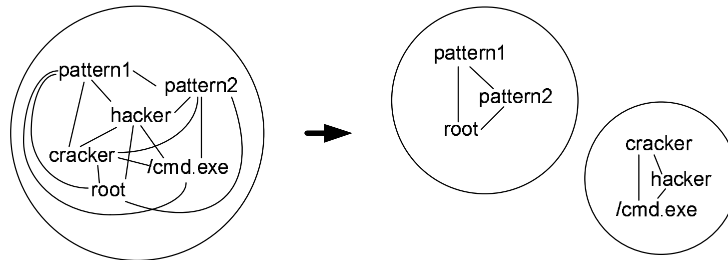

Fig. 2. Partitioned graph; by reducing the cut between the partitions we decrease the number of pipeline registers

Because intrusion detection requires a mix of case sensitive and insensitive alphabetic characters, numbers, punctuation, and hexadecimal-specified bytes, there is an interesting level of complexity. However, each string only contains a few dozen characters, and those characters tend to repeat across strings.

Using the min-cut heuristic, the patterns are partitioned n-ways such that the number of repeated characters within a partition is maximized, while the number of characters repeated between partitions is minimized. The system is then generated, composed of n pipelines, each with a minimum of bit lines. The value of n is determined from results; we have found n= 2-4 most effective for rulesets of less than 400 patterns. Conversely, for the 603 and 1000 pattern rulesets, the highest time performance is achieved with eight partitions. However, as the area increases moderately as the number of partitions increases, the area-time tradeoff must be considered. The tools, on average, can partition a ruleset to roughly 30 bits, or about the same amount of data routing in the 32 bit designs of [2, 3]. The matching units are least 8x smaller (32 down to 4 bits), and we have removed the control logic of KMP-style designs such as [14].

Our unary design utilizes a simple pipeline architecture for placing the appropriate bit lines in time. Because of the small number of total bit lines required (generally around 30) adding delay registers adds little area to the system design.

First, the tool partitions the patterns into several groups (Figure 2). Each group of patterns is handled by an independent pipeline to minimize the length of interconnects. We create a graph representation of the patterns to minimize the number of characters that have to be piped through each pipeline.

The graph creation strategy is as follows. We start with a collection of patterns, represented as nodes of a graph. Each pattern is composed of letters. Every node with a given letter is connected by an edge to every other node with that letter. We formalize this operation as follows:

$$S_k = \{a : a \in C \mid a \text{ appears in } k\} \quad (1)$$
$$V_R = \{p : p \in T\} \quad (2)$$
$$E_R = \{(k,l) : k, l \in T,\ k \neq l \text{ and } S_k \cap S_l \neq \emptyset\} \quad (3)$$

Graph creation before partitioning; a vertex V is added to graph R for each pattern p in the ruleset T and an edge E is added between any vertex-patterns that have a common character in the character class C

This produces a densely connected graph, almost 40,000 edges in a graph containing 361 vertices. Our objective is to partition the big graph into two or more smaller groups such that the number of edges between nodes within the group is maximized, and the number of edges between nodes in different groups is minimized. Each pipeline supplies data for a single group. By maximizing the edges internal to a group and minimizing edges outside the group which must be duplicated, we reduce the width of the pipeline registers and improve the usage of any given character within the pipeline. We utilize the METIS graph partitioning library [15].

One clear problem is that of large rulesets (>500 patterns). In these situations it is essentially impossible for a small number of partition to not each have the entire alphabet and common punctuation set. This reduces the effectiveness of the partitioning; however, if we add a weighting function the use of partitioning is advantageous into much larger rulesets. The weighting functions is as follows:

$$W_E = \sum_{i=1}^{min(|k|,|l|)} [(min(|k|,|l|) - i) \text{ if } (k(i) == l(i)) \text{ else } 0] \quad (4)$$

The weight W_E of the edge between k and l is equal to the number of characters $k(i)$ and $l(i)$ in the pattern, with the first character comparison weighted as the length of the shortest pattern. The added weight function causes patterns sharing character locality to be more likely to be grouped together.

The addition of the weighting function in Equation 4 allows the partitioning algorithms to more strongly put patterns with similar initial patterns of characters to be grouped together. The weighting function is weak enough to not force highly incompatible patterns together, but is strong enough to keep similar prefixes together. This becomes important in the tree approach, described next.

4 Tree-based Prefix Sharing

We have developed a tree-based strategy to find pattern prefixes that are shared among matching units in a partition. By sharing the matching information across

the patterns, the system can reduce redundant comparisons. This strategy allows for increased area efficiency, as hardware reuse is high. However, due to the increased routing complexity and high fanout of the blocks, it can increase the clock period. This approach is similar to the *trie* strategy utilized in [1], in which a collection of patterns is composed into a single regular expression. Their DFA implementation could not achieve high frequencies, though, limiting its usefulness. Our approach, utilizing a unary-encoded shift-and-compare architecture and allowing only prefix sharing and limited fanout, provides much higher performance. Beyond the strategic difference of the shift-and-compare architecture, our tree approach differs from the *trie* approach because in that it is customized for the 4-bit blocks of characters. This produces a lower depth tree as there is only a new branch for each block of four, making for a smaller architectural description generation problem. Moreover, the four character prefixes map to four decoded bits, fitting perfectly within a single Xilinx Virtex 4-bit lookup table.

Figure 3 illustrates the tree-based architecture. Each pattern (of length greater than 8) is composed of a first-level prefix and a second-level prefix. Each prefix is matched independently of the remainder of the pattern. After a single-clock delay, the two prefixes and the remainder of the pattern are combined together to produce the final matching information for the pattern. This is effective in reducing the area of the design because large sections of the rulesets share prefixes. The most common prefix is /scripts, where the first and second-level prefixes are used together. The 4-character prefix was determined to fit easily into the Virtex-style 4-bit lookup table, but it turns out that number is highly relevant to intrusion detection as well. Patterns with directory names such as /cgi-bin and /cgi-win can share the same first-level prefix, and then each have a few dozen patterns that share the -bin or -win second-level prefix.

	Number of Prefixes	
No. of Patterns	First Level	Second Level
204	83	126
361	204	297
602	270	421
1000	288	523

Table 1. Illustration of effectiveness of tree strategy for reducing redundant comparisons.

In Table 1, we show the various numbers of first and second-level prefixes for the various rulesets we utilized in our tests. Second-level prefixes are only counted as different within the same first-level prefix. For this table, we created our rulesets using the first n rules in the Nikto ruleset [7]. There is no intentional pre-processing before the tool flow. The table shows that, on average, between 2 and 3x redundancy can be eliminated through the use of the tree architecture. However, some of this efficiency is reduced due to the routing and higher fanout required because of the shared prefix matching units.

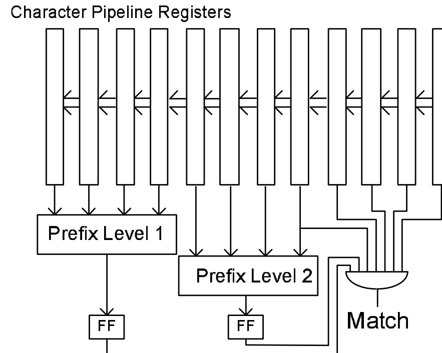

Fig. 3. Illustration of Tree-based hardware reuse strategy. Results from two appropriately delayed prefix blocks are delayed through registers and then combined with remaining suffix. The key to the efficiency of this architecture is that the prefix matches are reused, as well as the character pipeline stages

5 Performance Results for Tool-Generated Designs

This section will present results based on partitioning-only unary and tree architectures automatically by our tool. The results are based on ruleset of 204, 361, 602 and 1000 patterns, subsets of the Nikto ruleset of the Hogwash database [7].

We utilized the tool to generate system code for various numbers of partitions. Table 2 contains the system characteristics for partitioning-only unary designs, and Table 3 contains our results for the tree-based architecture. As our designs are much more efficient than other shift-and-compare architectures, the most important comparisons to make are between "1 Partition" (no partitioning) and the multiple partition cases. Clearly, there is an optimal number of partitions for each ruleset; this tends toward 2-3 below 400 patterns and toward 8 partitions for the 1000 pattern ruleset. The clock speed gained through partitioning can be as much as 20%, although this is at the cost of increased area. The tree approach produces greater increases in clock frequency, at a lower area cost. The 602 pattern ruleset shows the most dramatic improvements when using the tree approach, reducing area by almost 50% in some cases; the general improvement is roughly 30%. Curiously, the unpartitioned experiments actually show an increase in area due to the tree architecture, possible due to the increased fanout when large numbers of patterns are sharing the same pipeline.

Table 4 contains comparisons of our system-level design versus individual comparator-level designs from other researchers. We only compare against designs that are architecturally similar to a shift-and-compare discrete matcher, that is, where each pattern at some point asserts an individual signal after comparing against a sliding window of network data. We acknowledge that it is impossible to make fair comparisons without reimplementing all other designs. We have defined performance as throughput/area, rewarding small, fast designs. The synthesis tool for our designs is Synopsis Synplicity Pro 7.2 and the place and route tool is Xilinx ISE 5.2.03i. The target device is the Virtex II Pro XC2VP100 with -7 speed grade. We have done performance verification on the

		Number of Patterns in Ruleset			
	No. Partitions	204	361	602	1000
Clock Period	1	4.18	5.18	5.33	5.41
	2	4.45	4.50	5.60	5.17
	3	3.86	4.80	4.55	5.6
	4	3.99	4.24	5.06	5.22
	8	4.17	5.19	4.60	4.93
Area	1	800	1198	2466	4028
	2	957	1394	3117	4693
	3	1043	1604	3607	5001
	4	1107	1692	4264	5285
	8	2007	1891	5673	6123
Total chars in ruleset:		4518	8263	12325	19584
Characters per slice (min):		5.64	6.89	4.99	4.86

Table 2. Partitioning-only Unary Architecture: Clock period (ns) and area (slices) for various numbers of partitions and patterns sets

Xilinx ML-300 platform. This board contains a Virtex II Pro XC2VP7, a small device on the Virtex II spectrum. We have subsets of the database (as determined to fit on the device) and they execute correctly at the speeds documented in Table 2.

In Table 2 and 3, we see that the maximum system clock is between 200 and 250MHz for all designs. The system area increases as the number of partitions increases, but the clock frequency reaches a maximum at 3 and 4 partitions for sets under 400 rules and at 8 partitions for larger rulesets. Our clock speed, for an entire system, is in line with the fastest single-comparator designs of other research groups. On average, the tree architecture is smaller and faster than the partitioning-only architecture. In all cases the partitioned architectures (both tree and no-tree) are faster than the non-partitioned systems.

The smallest of designs in the published literature providing individual match signals is in [10], in which a state machine implements a Non-deterministic Finite Automata in hardware. That design occupies roughly 0.4 slice per character. Our tree design occupies roughly one slice per 5.5-7.1 characters, making it significantly more effective. While this approach is somewhat limited by only accepting 8 bits per cycle, the area efficiency allows smaller sets of patterns to be replicated on the device. This can increase throughput by allowing for parallel streams of individual 8-bit channels. For a single, high-throughput channel, the stream is duplicated, offset appropriately, and fed through duplicated matchers, allowing multiple bytes to be accepted in each cycle. The tool is capable of generating 4 and 8-byte systems as well (results are included in Table 4, descriptions of these architectures can be found in [13]).

After partitioning, each pattern within a given partition is written out, and a VHDL file is generated for each partition, as well as a system wrapper and testbench. The size of the VHDL files for the 361 ruleset total roughly 300kB in 9,000 lines, but synthesize to a minimum of 1200 slices. While the automation tools handle the system-level optimizations, the FPGA synthesis tools handle

		Number of Patterns in Ruleset			
	No. Partitions	204	361	602	1000
Clock Period	1	4.89	5.25	5.43	5.35
	2	4.18	4.27	4.8	4.22
	3	3.99	4.15	4.32	5.08
	4	4.1	4.1	4.54	4.69
	8	4.3	4.43	4.628	4.9
Area	1	773	1165	2726	4654
	2	729	1212	2946	3170
	3	931	1410	2210	5010
	4	1062	1345	2316	5460
	8	1222	1587	2874	6172
Total chars in ruleset:		4518	8263	12325	19584
Characters per slice (min):		6.19	7.09	5.577	6.17

Table 3. Tree Architecture: Clock period (ns) and area (slices) for various numbers of partitions and patterns sets

the low-level optimizations. During synthesis, the logic that is not required is pruned – if a character is only utilized in the shallow end of a pattern, it will not be carried to the deep end of the pipeline. If a character is only used by one pattern in the ruleset, and in a sense wastes resources by inclusion in the pipeline, pruning can at least minimize the effect on the rest of the design.

In the 361 pattern, 8263 character system, the design automation system can generate the character graph, partition, and create the final synthesizeable, optimized VHDL in less than 10 seconds on a desktop-class Pentium III 800MHz with 256 MB RAM. The 1000 pattern, 19584 character ruleset requires about 30 seconds. All of the code code except the partitioning tool is written in Perl, a runtime language. While Perl provides powerful text processing capabilities useful for processing the rulesets, it is not known as a high performance language. A production version of this tool would not be written in a scripting language. Regardless of the implementation, the automatic design tools occupy only a

Design	Frequency	Throughput	Unit Size	Performance
USC Unary	258 MHz	2.07 Gb/s	7.3	283
USC Unary (1 byte)	223 MHz	1.79 Gb/s	5.7	315
USC Unary (4 byte)	190 MHz	6.1 Gb/s	22.3	271
USC Unary (8 byte)	160 MHz	10.3 Gb/s	32.0	322
USC Unary (Tree)	250 Mhz	2.00 Gb/s	6.6	303
Los Alamos[4]	275 MHz	2.2 Gb/s	243	9.1
UCLA RDL [2]	100 MHz	3.2 Gb/s	11.4	280
GATech (NFA) [9]	218 MHz	7.0 Gb/s	50	140
U/Crete (FCCM) [3]	303 MHz	9.7 Gb/s	57	170

Table 4. Pattern size, average unit size for a 16 character pattern (in logic cells; one slice is two logic cells), and performance (in Mb/s/cell). Throughput is assumed to be constant over variations in pattern size

small fraction of the total hardware development time, as the place and route of the design to FPGA takes much longer, roughly ten minutes to complete for the 361 pattern, 8263 character design.

6 Conclusion

This paper has discussed a tool for system-level optimization using graph-based partitioning and tree-based matching of large intrusion detection pattern databases. By optimizing at a system level and considering an entire set of patterns instead of individual string matching units, our tools allow more efficient communication and extensive reuse of hardware components for dramatic increases in area-time performance. Through preprocessing, our tool automatically generates highly area-efficient designs with competitive clock frequencies.

We release the collection of tools used in this paper to the community at `http://halcyon.usc.edu/~zbaker/idstools`

References

1. Hutchings, B.L., Franklin, R., Carver, D.: Assisting Network Intrusion Detection with Reconfigurable Hardware. In: Proceedings of FCCM '02. (2002)
2. Cho, Y., Mangione-Smith, W.H.: Deep Packet Filter with Dedicated Logic and Read Only Memories. In: The Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04). (2004)
3. Sourdis, I., Pnevmatikatos, D.: A Methodology for the Synthesis of Efficient Intrusion Detection Systems on FPGAs. In: The Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04). (2004)
4. Gokhale, M., Dubois, D., Dubois, A., Boorman, M., Poole, S., Hogsett, V.: Granidt: Towards Gigabit Rate Network Intrusion Detection. In: Proceedings of FPL '02. (2002)
5. Moscola, J., Lockwood, J., Loui, R.P., Pachos, M.: Implementation of a Content-Scanning Module for an Internet Firewall. In: Proceedings of FCCM '03. (2003)
6. Sourcefire: Snort: The Open Source Network Intrusion Detection System. `http://www.snort.org` (2003)
7. Hogwash Intrusion Detection System: (2004) `http://hogwash.sourceforge.net/`.
8. Global Velocity: (2004) `http://www.globalvelocity.info/`.
9. Clark, C.R., Schimmel, D.E.: Scalable Parallel Pattern Matching on High Speed Networks. In: The Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04). (2003)
10. Clark, C.R., Schimmel, D.E.: Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In: Proceedings of FPL '03. (2003)
11. Cho, Y.H., Navab, S., Mangione-Smith, W.H.: Specialized Hardware for Deep Network Packet Filtering. In: Proceedings FPL '02. (2002)
12. Sourdis, I., Pnevmatikatos, D.: Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System. In: Proceedings of FPL '03. (2003)
13. Baker, Z.K., Prasanna, V.K.: A Methodology for the Synthesis of Efficient Intrusion Detection Systems on FPGAs. Accepted for publication at FCCM '04 (2004)
14. Baker, Z.K., Prasanna, V.K.: Time and Area Efficient Pattern Matching on FPGAs. In: Proceedings of FPGA '04. (2004)
15. Karypis, G., Aggarwal, R., Schloegel, K., Kumar, V., Shekhar, S.: METIS Family of Multilevel Partitioning Algorithms (2004)