A Codesigned On-Chip Logic Minimizer

Roman Lysecky, Frank Vahid*
Department of Computer Science and Engineering
University of California, Riverside
{rlysecky, vahid}@cs.ucr.edu
*Also with the Center for Embedded Computer Systems at UC Irvine

Abstract

Boolean logic minimization is traditionally used in logic synthesis tools running on powerful desktop computers. However, logic minimization has recently been proposed for dynamic use in embedded systems, including network route table reduction, network access control list table reduction, and dynamic hardware/software partitioning. These new uses require logic minimization to run dynamically as part of an embedded system's active operation. Performing such dynamic logic minimization on-chip greatly reduces system complexity and security versus an approach that involves communication with a desktop logic minimizer. An on-chip minimizer must be exceptionally lean yet yield good enough results. Previous software-only on-chip minimizer results have been good, but we show that a codesigned minimizer can be much better, executing nearly 8 times faster and consuming nearly 60% less energy, while yielding identical results.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms

Algorithms, Performance.

Keywords

Logic minimization, dynamic optimization, on-chip logic minimization, on-chip synthesis, system-on-a-chip, embedded systems, hardware/software codesign, embedded CAD.

1. Introduction

On-chip logic minimization is becoming increasingly important in applications requiring dynamic optimizations. Unfortunately, most logic minimization algorithms have been developed with the expectation the algorithms will be run on a desktop workstation. The use of these minimization algorithms for performing dynamic optimizations within an embedded system is limited. While the applications of on-chip logic minimization are just recently emerging, logic minimization is already useful in performing dynamic hardware/software partitioning, dynamically reducing network routing table size, and dynamically reducing network access control lists.

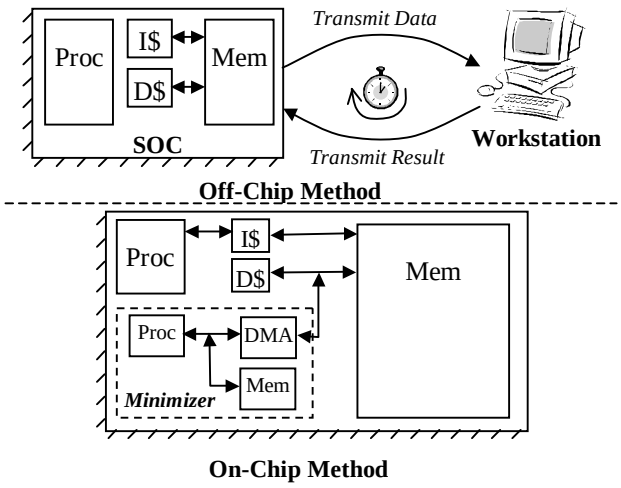
CODES+ISSS'03, October 1-3, 2003, Newport Beach, California, USA.

In [12], we presented a first approach to dynamic hardware/software partitioning. In this approach, the dynamic partitioning system monitors an executing application, determines the critical loops of the application, and executes these loops in hardware using on-chip configurable logic. Such an approach requires several tools to convert the software description of the loop to hardware, including decompilation, logic synthesis, technology mapping, and place and route. During the logic synthesis phase, the dynamic hardware/software partitioning system uses two-level logic minimization in an iterative process to optimize the hardware circuit, requiring an on-chip logic minimizer.

Logic minimization is also useful for performing optimization in other applications not related to synthesis of hardware circuits. Such applications include IP (Internet Protocol) routing table reduction and network access control list (ACL) reduction. Network routers route an incoming IP packet to its destination by determining the packet's next hop. The router compares the packet's destination IP with the router's routing table and uses the longest prefix match to select the packet's destination port. While simple lookup schemes work for small routers, large network routers with ten of thousands of routing table entries can require long lookup times. Longest prefix matching can be directly mapped to Ternary CAMs [9]. However, TCAMs require large hardware resources and tend to have high power consumption. Liu developed a method for reducing the routing table size using two-level logic minimization and hence reducing the cost of using TCAMs [7].

Logic minimization is also helpful in reducing the size of a network router's access control list. Most commercially available

Figure 1: Comparison of off-chip and on-chip logic minimization methods.



network routers have ACLs that allow the router to accept or deny incoming IP packets. The router uses information from the incoming IP packet to search the ACL for the first matching entry and takes the associated action of either permitting the packet or denying the packet. Large ACLs often have thousands of entries, making sequential lookup infeasible. Furthermore, hardware-based parallel lookup approaches are limited because the approach must preserve the ordering of the ACL entries. In [8], we developed an approach using TCAMs to efficiently search ACLs, involving an optimization technique using two-level logic minimization to reduce the size of the ACL.

Dynamic hardware/software partitioning, IP routing table reduction, and ACL reduction require dynamic execution of logic minimization. Figure 1 highlights two possible methods for performing logic minimization dynamically. In the off-chip method, the application transfers the required data to a logic minimizer executing on a desktop workstation accessible through a communication link such as Ethernet. The workstation optimizes the data and transmits the results back to the application. While an off-chip method has the benefit of using a powerful workstation-based minimizer, the communication overhead can greatly slow the minimization process, especially when minimization is applied frequently. The method also has drawbacks related to ensuring the workstation is always available, and reducing security risks (ACL lists are heavily guarded by most companies, for example).

Alternatively, we can perform logic minimization on-chip by adding an on-chip minimizer to the chip itself. One approach to this method executes the optimizer as an additional task that shares the same processing resources as the application itself. Another approach to the on-chip method implements the minimizer using a separate embedded processor/memory system with direct access to the main processor's memory. Both on-chip approaches benefit from reducing or eliminating data transfers between the application and the logic minimizer, and they eliminate the complexity and security issues of the off-chip approach.

In either on-chip minimization approach, the logic minimizer will have limited processing resources, in terms of both processor speed and available instruction and data memory. These limitations require the need for a lean minimizer designed to execute on-chip. However, a lean minimizer must still provide "good enough" results within a reasonable time. We developed a lean software minimizer, ROCM (Riverside On-Chip Minimizer), designed to execute on an ARM7 processor [2], which is a low-cost lean 32-bit embedded processor. While the use of ROCM to perform IP routing table reduction, ACL reduction, and dynamic hardware/software partitioning provided good results with very low memory requirements, these benefits came at the cost of slower execution speed and additional energy consumption require by the minimizer.

In this paper, we present a codesigned on-chip minimizer that improves performance and reduces energy consumption. We started with customized versions of ROCM, called ROCM-32 and ROCM-128, customized for applications with fixed input sizes of 32 bits and 128 bits, respectively. In our approach, we first modified the data structures and algorithms used by ROCM to make ROCM more amenable to hardware/software partitioning. We then isolated the critical computational kernels and partitioned these functions/loops to a minimization coprocessor. Our codesigned on-chip minimizer results in better execution time and reduced energy.

2. Previous work

The general two-level logic minimization problem for heuristic minimizers can be stated as:

> *Given the inputs F (cover of the on-set) and D (cover of the don't care set) of an incompletely specified logic function, determine a cover of F that is minimal, where a minimal cover of F is a cover that is not a proper superset of any other cover of the function.*

We designed our original two-level logic minimization algorithm, ROCM, employing techniques used by both Espresso-II [3] and Presto [12], which form the heart of many popular commercial synthesizers today. The ROCM algorithm uses a simple heuristic approach with a single expand phase implemented with the main goal of very small memory usage. First, ROCM orders the implicants according to decreasing implicant size, under the assumption that larger implicants are more likely to cover other implicants and less likely to be covered by other cubes [3]. ROCM uses an iterative expansion process that attempts to expand each implicant and determines if each expansion is valid. While the use of the off-set for determining the validity of expanded implicants, as is done by Espresso-II, yields a very efficient algorithm, the size of the off-set can be very large. We therefore chose not to compute the off-set, and we instead employ a tautology-based approach similar to that used by Presto. Using this simplified methodology, ROCM required on average an order of magnitude less code size and only one third the data memory of Espresso-II. ROCM's low memory requirements came at the expense of a mere 2% decrease in minimization quality. ROCM's execution time (ignoring communication time) was 13% longer, but when applied incrementally (as typical for the network applications) was 10 times faster, and when executing on an ARM7 was still 2-5 times faster than Espresso-II on a 500 MHz Sun Ultra60 workstation.

In embedded systems, customizing an algorithm for a particular application is often beneficial. A customized version of our logic minimizer would require less memory and reduce dynamic memory allocation while improving performance. Therefore, in our original work, we created customized versions of ROCM, called ROCM-32 and ROCM-128, optimized for applications with known input size [8]. For example, ROCM-32 is optimized for routing table reduction applications where the input size is 32-bits. ROCM-32 required an average of 11% less data memory and 37% less execution time than the non-customized version.

Cong et al [4][5] sped up two-level logic minimization by utilizing an FPGA to speedup the tautology checking algorithm of Espresso-II. In their work, the hardware coprocessor performs tautology checking for functions of eight or fewer variables. Their implementation using an FPGA achieved an average speedup of 1.36, with a maximum speedup of 2.94. The main limitation of their approach is the use of a minterm evaluator for which 2^k terms must be evaluated, where k is the number of input variables. The hardware required for a larger number of

Table 1: ROCM-32 profiling results for IP router table reduction.

Function/ Loop	Total Instr.	Loop Instr.	Loop Time%	Loop Size%	Speedup Bound
DoesInter	15401	34	42.7%	0.22%	1.7
SetLit	15401	23	5.7%	0.15%	1.1
GetLit	15401	8	8.8%	0.05%	1.1
IsCov	15401	67	3.4%	0.44%	1.0
Tautology.1	15401	67	1.7%	0.37%	1.0
Cofactor.1	15401	57	28.5%	0.44%	1.4
Overall	**15401**	**256**	**90.7%**	**1.68%**	**10.8**

variables quickly becomes infeasible using their coprocessor design.

In designing our codesign on-chip logic minimizer, we can exploit the customization of ROCM to the input size of the intended application. The known input size of the minimization algorithms allows for better partitioning and small hardware requirements. A small, embedded processor is still required as there is much dynamic memory allocation and control logic, which is more feasible to implement in software than hardware. We will design the hardware coprocessor to efficiently implement the critical computational kernels within our minimization algorithms.

3. Codesigned On-Chip Logic Minimizer

We performed an initial profiling of the customized ROCM-32 and ROCM-128 programs to determine the critical kernels of those logic minimizers. To profile the programs, we used a SimpleScalar [11] simulator, ported for the ARM family of processors, and extended to produce an instruction trace. We then used a loop analysis tool [15] to process the instruction trace along with the disassembled binary executable to determine the critical functions, loops, and sub-loops. Our initial profiling of the logic minimizers revealed good potential speedups. However, we noticed some initial limitations resulting from our algorithms' implementation and the organization of data structures. Our early investigation identified six critical kernels from within ROCM-32 and ROCM-128 that could be potentially implemented in hardware. However, we initially designed our algorithms using dynamic memory allocation within the critical software loops. We modified these algorithms to remove the dynamic memory allocation from within the critical loops. While we were able to rewrite the algorithms to eliminate the memory allocation from the loops, we added additional code before the critical loop to handle the memory allocation, resulting in 3% slower execution. Fortunately, the degradation in execution speed is easily overcome after partitioning our minimizer.

Additionally, we performed another simple modification to the logic minimization code to improve the potential speedup we could achieve using a hardware coprocessor. The basic underlying data structure used for storing implicants contains many different data items. While these data items are used for various purposes throughout the execution of our logic minimizer, many of these data items are not needed during the critical portions of code. However, in the original C code, the pertinent data items were interleaved with the non-essential data items throughout the data structure. Using a compiler that byte-aligns data structures, the data items that we are interested in will reside at multiple memory locations. Therefore, the original data structure organization would require more memory accesses by the hardware coprocessor during execution. Therefore, we reorganized the data structures to ensure the required data items were located in sequential memory locations as tightly packed as possible.

Table 2: ROCM-128 profiling results for ACL reduction.

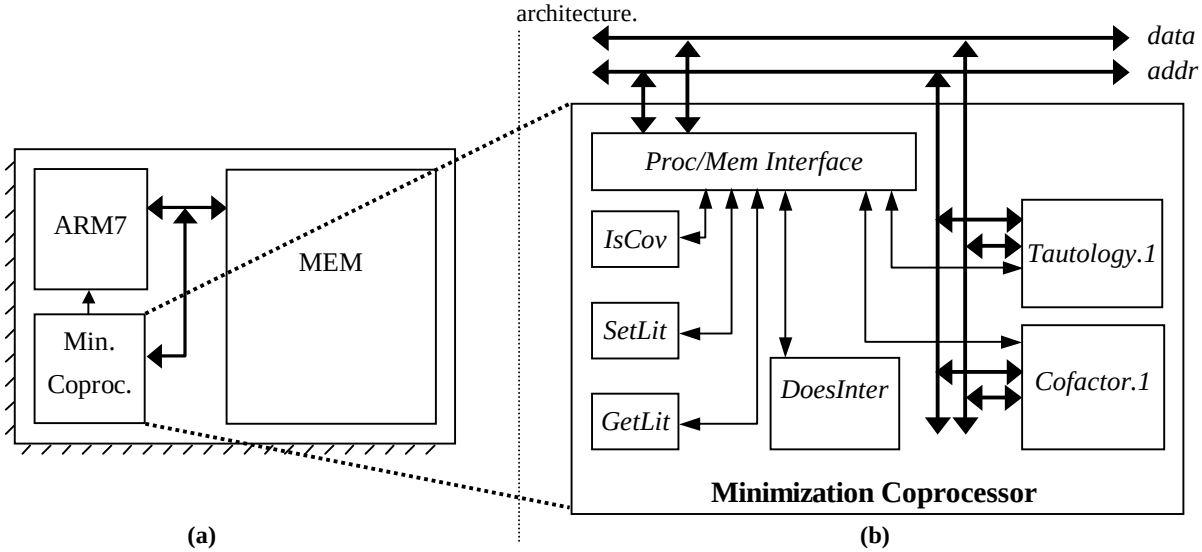
Function/ Loop	Total Instr.	Loop Instr.	Loop Time%	Loop Size%	Speedup Bound
DoesInter	15518	51	21.3%	0.33%	1.3
SetLit	15518	27	21.3%	0.17%	1.3
GetLit	15518	12	25.9%	0.08%	1.4
IsCov	15518	83	0.5%	0.54%	1.0
Tautology.1	15518	204	11.0%	1.32%	1.1
Cofactor.1	15518	59	12.8%	0.38%	1.1
Overall	**15518**	**436**	**92.8%**	**2.81%**	**13.9**

This necessary rewriting of the algorithms and data structures for more efficient hardware execution is something often overlooked in codesign research. The rewriting was quite fundamental and likely could not be automated easily using existing compiler or synthesis transformation methods. Vahid et al [6] observed this phenomenon in several other applications. The necessary rewriting has implications for top-down codesign approaches that seek to automatically partition a single specification among hardware and software.

After we modified our logic minimization tools, we again profiled ROCM-32 for IP routing table reduction and profiled ROCM-128 for network ACL reduction. Table 1 displays the profiling results for ROCM-32, and Table 2 displays the profiling results for ROCM-128, listing the six critical kernels that we identified. *Function/Loop* identifies the critical kernels. *TotalInstr* is the total number of instructions of the application. *LoopInstr* is the number of instructions within the critical kernel. *LoopTime%* is the percent of overall execution time spent within the critical kernel. *LoopSize%* is the percent of overall code size corresponding to the critical kernel. *SpeedupBound* is the maximum speedup possible assuming the critical kernel is executed in zero time. For ROCM-32, the six critical functions/loops account for over 90% of the execution time but less than 2% of the overall code. Similarly, ROCM-128 accounts for over 90% of the execution time and only 3% of the code. The six critical kernels include *DoesInter*, *SetLit*, *GetLit*, *IsCov*, *Tautology.1*, and *Cofactor.1*. *DoesInter* determines if two implicants intersect. *SetLit* sets a literal within an implicant to the specified value. *GetLit* retrieves a specific literal from within an implicant. *IsCov* determines if an implicant is covered by another implicant. *Tautology.1* is a loop that handles special cases for determining if the specified function is a tautology. Finally, *Cofactor.1* is the main loop that determines the cofactor with respect to a specified implicant.

Figure 2(a) shows the overall architecture of our codesigned minimizer, which consists of an ARM7 processor coupled with a

Figure 2: Codesigned on-chip logic minimizer architecture: (a) overall codesigned minimizer architecture, (b) detailed co-processor architecture.



small memory and the minimization coprocessor. Additionally, the coprocessor has direct access to memory. We implemented all six critical kernels within the coprocessor. We modified the minimization software running on the ARM7 to activate the coprocessor instead of executing the original code. During the coprocessor activation, the software transfers the required initialization information along with an encoded value indicating which of the six kernels is to be executed. After activating the coprocessor, we place the ARM7 in a low power mode until the coprocessor wakes up the processor using an interrupt, signaling completion.

Figure 2(b) shows a more detailed architecture of the minimization coprocessor. The *Proc/Mem Interface* is responsible for communication with the processor and internal hardware components. The *Proc/Mem Interface* also handles all memory accesses for *DoesInter*, *GetLit*, *SetLit*, and *IsCov*, as all four of these hardware components are implemented as combinational logic consisting of simple logic gates, 2-bit muxes, and shifters. On the other hand, *Tautology.1* and *Cofactor.1* perform many memory accesses, as both algorithms must traverse an array of implicants during execution. Therefore, we implemented *Tautology.1* and *Cofactor.1* as separate hardware components that receive the initialization data from the *Proc/Mem Interface* but during execution have direct access to memory. While we can implement the internal computation required by *Cofactor.1* using simple logic gates, adders, and shifters, *Tautlogy.1* also requires the use of a 32-bit multiplier.

We created two codesigned on-chip logic minimizers for the two distinct input sizes needed for our applications. The codesigned version of ROCM-32 will be referred to as CD-ROCM-32, and the codesigned version of ROCM-128 will be referred to as CD-ROCM-128.

4. Results

While our main requirements for an on-chip logic minimizer is small instruction and data memory requirements, the execution time, required hardware area, and power consumption are also important. In our original approach, we only required a small ARM7 processor to execute our logic minimization software. The addition of an ARM7 dedicated to executing our software minimizer would require roughly 100,000 gates including caches and consume 49 mW of power executing at 75 MHz. Such an addition is small compared with current chip capacities of several million gates; we saw a router chip 3 years ago with 36 ARM processors. Nevertheless, we can eliminate the need for an additional processor by sharing a single processor with the main application, if the main application's time constraints allow for this sharing. However, our codesigned on-chip logic minimizer requires the use of dedicated hardware resources.

We implemented the coprocessor for CD-ROCM-32 and CD-

Table 3: Codesigned on-chip logic minimizer speedup and energy reduction for IP routing table reduction, ACL reduction, and dynamic HW/SW partitioning logic synthesis compared with ROCM-32 and ROCM-128.

Codesigned Minimizer	Performance (s)					Energy (mJ)				
	Orig. SW Time	SW Loop Time	HW Loop Time	HW/SW Time	Speedup	Orig. SW Energy	SW Energy	HW Energy	Total Energy	Energy Savings
CD-ROCM-32 (IP)	1.64	1.49	0.06	0.22	7.6	80.1	8.2	6.3	14.6	81.8%
CD-ROCM-128 (ACL)	1.53	1.42	0.08	0.19	7.9	74.7	6.4	32.3	38.8	48.1%
CD-ROCM-128 (LogSyn)	0.054	0.050	0.003	0.007	7.9	2.61	0.22	1.14	1.36	47.8%
				Average:	**7.8**				**Average:**	**59.2%**

Table 4: Execution time (*seconds*) of ROCM, ROCM-32, and CD-ROCM-32 for IP routing table reduction, and speedup of CD-ROCM-32 versus ROCM executing on a 75MHz ARM7.

Example	ROCM	ROCM-32	CD-ROCM-32	Speedup
MaeWest	2214	1328	175	12.7
AADS	2341	1419	187	12.5
Paix	271	199	26	10.4
PacBell	2666	1581	208	12.8
			Average:	**12.1**

ROCM-128 using VHDL to determine the required hardware area and power consumption. We synthesized both coprocessors using Synopsys Deign Compiler [14] using the UMC 0.18µm technology library provided by Artisan Components [1]. Furthermore, we determined the power consumption of both coprocessor using Design Compiler and gate-level simulations. The CD-ROCM-32 coprocessor required roughly 19,000 gates and has a power consumption of 100 mW when active. The CD-ROCM-128 minimizer required roughly 32,000 gates with a power consumption of 388 mW. Both coprocessors execute at 200 MHz.

Table 3 highlights the speedup of our codesigned minimizers for IP routing table reduction, ACL reduction, and logic synthesis optimization with dynamic hardware/software partitioning. For IP routing table reduction, we compared ROCM-32 with our codesigned CD-ROCM-32, and for ACL reduction and dynamic hardware/software partitioning, we compared ROCM-128 with our codesigned CD-ROCM-128. *Orig. SW Time* is the time required by our software only minimizer. *SW Loop Time* is the total time required by the six critical kernels when implemented in hardware. *HW Loop Time* is the time required for those critical kernels when executed using our hardware coprocessor. *HW/SW Time* is the total time for required by our codesigned minimizer. *Speedup* is the overall speedup of our codesigned minimizer compared to the all software version. Our overall speedup for CD-ROCM-32 over an all software approach is 7.6, and our overall speedup of CD-ROCM-128 is 7.9.

Table 3 also provides the energy reduction of our codesigned minimizers over the software based ROCM-32 and ROCM-128. *Orig. SW Energy* is the energy required by our original software minimization tools. *SW Energy* is the energy consumed by the ARM7 to execute the software for our codesigned logic minimizers. *HW Energy* is the energy required by our minimization coprocessors. *Total Energy* is the total energy required by our codesigned on-chip logic minimizer. Finally, *Energy Savings* is the percent energy reduction of our codesigned logic minimizers compared with ROCM-32 and ROCM-128. We calculated the energy required for ROCM-32, ROCM-128, and our codesigned minimizers we used the following set of equations:

$$E_{total} = E_{ARM} + E_{HW}$$

$$E_{ARM} = P_{ARM(idle)} \times t_{idle(ARM)} + P_{ARM(active)} \times t_{active(ARM)}$$

$$E_{HW} = P_{HW} \times t_{active(ARM)} + P_{static} \times t_{total}$$

Table 5: Execution time (*seconds*) of ROCM, ROCM-128, and CD-ROCM-128 for ACL reduction, and speedup of CD-ROCM-128 versus ROCM executing on a 75MHz ARM7.

Example	ROCM	ROCM-128	CD-ROCM-128	Speedup
Bad	15.63	8.22	1.04	15.0
Typ1	6.87	5.89	0.72	9.5
Typ2	6.15	2.62	0.33	18.5
Long	1572.44	686.81	86.94	18.1
Univ	17.98	86.94	1.16	15.6
			Average:	**15.3**

Although the minimization coprocessors required up to an order of magnitude more power than the ARM7 processor, the overall energy reduction of our codesigned minimizers is on average 59.2%. Furthermore, CD-ROCM-32 requires over 81% less energy than ROCM-32, while CD-ROCM-128 requires roughly 48% less energy than ROCM-128.

We also analyzed the performance of our codesigned on-chip logic minimizers compared to our original on-chip minimizer, ROCM, and the customized version ROCM-32 and ROCM-128. Table 4 presents overall execution times of ROCM, ROCM-32, and CD-ROCM-32 in performing routing table reduction using data from for four large network routers, MaeWest, AADS, Paix, and PacBell, from data available at [10]. Furthermore, Table 5 presents overall execution time for performing ACL reduction of five access control lists ranging in size from 99 entries to over 3000 entries. For performing IP routing table reduction, our codesigned on-chip minimizer CD-ROCM-32 has a maximum speedup of 12.8 and an average speedup of 12.1 compared to ROCM. Furthermore, although logic minimization is executing on-chip using a 75 MHz ARM7 processor coupled with a 200 MHz coprocessor, CD-ROCM-32 is on average 1.2 times faster than Espresso-II executing on a 500 MHz Sun Ultra60 workstation for performing IP routing table reduction. For performing ACL reduction, CD-ROCM-128 has a maximum speedup of 18.5 and an average speedup of 15.3 compared to our initial on-chip logic minimizer.

5. Conclusions

On-chip logic minimization requires a fast and lean minimizer component producing "good enough" results. We have developed such a component through hardware/software partitioning, achieving speedups of nearly 8x compared to an earlier software-only minimizer component, while using a very reasonable number of extra gates. The codesigned component also reduces energy by more than half. During our codesign process, we needed to modify the underlying data structures and algorithms of ROCM, which may imply new directions of hardware/software partitioning research. The feasibility of fast and lean on-chip logic minimization may open up many new applications of logic minimization, beyond the networking and dynamic partitioning applications we described, that were previously not considered feasible due to the complexity of off-chip logic minimization.

6. Acknowledgements

This work was supported in part by the National Science Foundation (CCR-0203829), the Semiconductor Research Corporation, and a Department of Education GAANN fellowship.

7. References

[1] Artisan Components. http://www.artisan.com, 2003.

[2] Advanced RISC Machines Ltd. ARM7. http://www.arm.com/armtech/ARM7_Thumb/, 2002.

[3] Brayton, R., et al. Logic Minimization Algorithms for VLSI Synthesis. Kluwer Academic Publishers, Boston, MA, 1984.

[4] Cong, J., J. Peck. On Acceleration on the Check Tautology Logic Synthesis Algorithm using an FPGA-based Reconfigurable Coprocessor. Proc. Field-Programmable Custom Computing Machines (FCCM), 1997.

[5] Cong, J., J. Peck. On Acceleration on the Check Tautology Logic Synthesis Algorithm using an FPGA-based Reconfigurable Coprocessor. Technical Report TR-970010, University of California, Los Angeles, 1997.

[6] Grattan, B., G. Stitt and F. Vahid. Codesign Extended Applications. IEEE/ACM Int. Symposium on Hardware/Software Codesign, May 2002, pp. 1-6.

[7] Liu, H. Routing Table Compaction in Ternary-CAM. IEEE Micro, pp. 58-64, Jan/Feb 2002.

[8] Lysecky, R., F. Vahid. On-Chip Logic Minimization. Proc. 40th Design Automation Conference, 2003.

[9] McAuley, A. P. Francis. Fast Router Table Lookup Using CAMs. Proc. Infocom, Vol. 3, pp. 1382-91, 1993.

[10] Merit Network, Inc. Internet Routing Table Statistics, http://www.merit.edu/ipma/routing_table/, 2002.

[11] SimpleScalar LLC. http://www.simplescalar.com, 2003.

[12] Stitt, G., R. Lysecky, F. Vahid. Dynamic Hardware/Software Partitioning: A First Approach. Proc. 40th Design Automation Conference, 2003.

[13] Svoboda, A., D.E. White. Advanced Logical Circuit Design Techniques. Garland Press, New York, 1979.

[14] Synopsys, Inc. Design Compiler. http://www.synopsys.com, 2003.

[15] Villarreal, J., R. Lysecky, S. Cotterell, F. Vahid. Loop Analysis of Embedded Applications. UC Riverside Technical Report UCR-CSE-01-03, 2001.