Exploration of energy efficient memory organisations for dynamic media applications using system scenarios

Iason Filippopoulos[1], Francky Catthoor[2], and Per Gunnar Kjeldsberg[1]

[1] NTNU, Trondheim, Norway
[2] IMEC, Leuven, Belgium

Abstract. System scenario methodologies propose the use of different scenarios, e.g., different platform configurations, in order to exploit variations in computational and memory needs during the lifetime of an application. In this paper several extensions are proposed for a system scenario based methodology with a focus on improving memory organisation. The conventional methodology targets mostly execution time while this work aims at including memory costs into the exploration. The effectiveness of the proposed extensions is demonstrated and tested using two real applications, which are dynamic and suitable for execution on modern embedded systems. Reductions in memory energy consumption of 40 to 70% is shown.

1 Introduction

Modern embedded systems are becoming more and more powerful as the semiconductor processing technique keep increasing the number of transistors on a single chip. Consequentially, demanding applications, such as medical signal processing and streaming applications, can be executed on these devices [1]. On the other hand, the desired performance has to be delivered with the minimum power consumption due to limited amount of power offered in mobile devices [2]. System scenario methodologies propose the use of different platform configurations in order to exploit variations in computational and memory needs often seen during the lifetime of such applications [2]. A platform can, e.g., be configured through frequency/voltage scaling or turning certain processing units on or off. In this work a reconfigurable memory platform is employed in order to study the effectiveness of a memory-aware system scenario methodology.

As shown in [3] memory contributes around 40% to the overall power consumption in general purpose systems. Especially for embedded systems, the memory subsystem accounts for up to 50% of the overall energy consumption [4] and the cycle-accurate simulator presented in [5] estimates that the energy expenditures in the memory subsystem range from 35% up to 65% for different architectures. According to [2], conventional allocation and mapping of data done by regular compilers is suboptimal. Performance loss is caused by stalls for fetching data and data conflicts for different tasks, due to the limited size of

memory and the competition between tasks. In addition, modern applications exhibit more and more dynamism. This gives a strong motivation for study and optimization of memory organisation in embedded devices with strongly dynamic application behaviour.

This paper is organized as follows. Section 2 surveys related work on system level exploration and on system scenario methodologies. Section 3 presents the chosen methodology and our novel extensions to make it applicable in a memory organisation study. In Section 4 the target platform is described while the demonstrator applications are presented in Section 5. Results of applying the described methodology to the targeted applications are shown in Section 6, while conclusions are drawn in Section 7. The main contribution of the current work is the proposal of extensions to the existing system scenario methodology, in order to take into account memory costs while performing the design exploration.

2 Related Work and Contribution Discussion

Many papers have focused on memory related optimisations, also in the presence of a partitioned and distributed memory organisation with memory blocks of different sizes (e.g. [6], [7], [8], [9]). However, they do not incorporate sufficient support for very dynamically behaving application codes. System scenarios allow to alleviate this bottleneck and to handle such dynamic behaviour. An overview of work on system scenario methodologies and their application are presented in [10]. So far memory organisation is rarely considered and not fully analysed. Furthermore, the majority of the published work focus on control variables for scenario prediction and selection. They can take a relatively small set of different values that can be fully explored. However, the use of data variables [11] is required for these tasks by many dynamic systems with value ranges that make full exploration impossible.

Authors in [12] present a technique to optimise memory accesses for input data dependent applications by duplicating and optimising the code for different execution paths of a control flow graph (CFG). One path or a group of paths in a CFG form a scenario and its memory accesses are optimized using global loop transformations (GLT). Apart from if-statement evaluations that define different execution paths, they extend their technique to include while loops with variable trip count in [13]. A heuristic to perform efficient grouping of execution paths for scenario creation is analysed in [14]. However, our work extends the existing solutions towards exploiting the presence of a distributed memory organisation with reconfiguration possibilities.

Reconfigurable hardware for embedded systems, including the memory architecture, is a topic of active research. An extensive overview of current approaches is found in [15]. The approach presented in this paper differentiates by focusing on the data-to-memory partitioning aspects in the presence of a platform with dynamically configurable memory blocks.

3 Extended System Scenario Methodology

The system scenario methodology is based on the observation that the workload of most systems varies significantly during their lifetime due to dynamic variation of computational and memory needs in the application code. Most of the existing design methods define the worst case execution time (WCET) of the most demanding task and tune the system in order to meet its needs [2]. Obviously, this approach leads to wasted time and memory area for tasks with lower execution time and memory requirements, since those tasks could meet their needs using fewer resources and consequentially consuming less energy.

In contrast, designing with scenarios is workload adaptive and offers different configurations of the platform and the freedom of switching to the most efficient scenario at run-time. In contrast to use case scenario approaches in which scenarios are generated based on a user's behaviour, the system scenario methodology focuses on behaviour of the system to generate scenarios. A system scenario is a configuration of the system that combines similar run-time situations (RTSs). An RTS consists of a running instance of a task and its corresponding cost (e.g. energy consumption) and one complete run of the application on the target platform represents a sequence of RTSs [11]. The system is configured to meet the cost requirements of an RTS by choosing the appropriate scenario, which is the one that satisfies the requirements using minimal power.

In the following subsections, the different steps of the system scenario methodology is outlined, with emphasis on the new extensions that make it possible to take memory into account.

3.1 General description of system scenario methodology

The scenario methodology follows a two stage exploration, namely design-time and run-time stages, as described in [10]. The two stage exploration is chosen because it reduces run-time overhead while preserving an important degree of freedom for run-time configuration [2]. In more detail, the application is analysed at design-time and different execution paths and variations in processing and memory demands are identified. This procedure, which is time consuming and as a result can be performed only during the design phase, will result in a grey-box model representation of the application. The grey-box model hides all static and deterministic parts of the application, by providing only related costs for those, and keeps parts of the application code that are non-deterministic available to the system designer [16]. This way, more focus can be given to parts of the application that have impact on the cost variations so that different execution decisions can be studied. Those different options are made available to the system designer and each decision could be either fixed at design time or forwarded for dynamic evaluation at run-time.

3.2 Design-time Profiling

Application profiling is performed at design-time and consists of an analysis of the target application during its lifetime and for a wide range of inputs. The

Fig. 1. Profiling results based on application code and input data.

analysis focuses on execution time for a reference processor in the conventional system scenario methodology, but in our case the cost will be memory size. This cost metric is chosen, because several applications allocate and deallocate memory space during their execution. Also, data reuse behavior and decisions made by the system designer about the size of sets of data to be copied to different units in the memory hierarchy (i.e., copy-candidates [17]) strongly influences the memory size. Energy consumption and access time of a memory unit is affected by its size [18]. Together with access pattern information, the memory size is hence an important metric.

The flow of the profiling stage is depicted in Fig. 1 and consists of running the application code with suitable input data often found in a database, in order to produce profiling results. This reveals parts of the application code with high memory activity and with varying memory access intensity, which possibly depends on input data. Because of this behaviour, a static study of the application code alone is insufficient since the target applications for this methodology have non-deterministic behaviour that is driven by input. Choosing an extensive and accurate database is vital and will heavily influence and steer the designer's decisions in later steps.

Given code and database as inputs, profiling will show memory usage during execution time by running the application using the whole database as an input. Results provided to the designer include complete information about allocated memory size values together with the number of occurrences and duration for each of these memory size values. Moreover, correlation between input data values and the resulting memory behaviour can possibly be observed. This information is forwarded to the next stage.

There are several ways that profiling can be performed. Current program monitoring software cannot offer the needed level of detail. Debuggers and memory tools, such as Valgrind [19], or direct hard-coded profiling are preferable methods, because of their higher accuracy.

3.3 Design-time Scenario Identification and Prediction

Scenario generation is also performed during design time and is the procedure of clustering profiled information into groups with similar characteristics. More precisely, cost points are clustered in groups based on their distance in cost axis and the frequency of their occurrence. Clustering is necessary, because it will be extremely costly to have a different scenario for every possible situation. Switching cost, which is the cost of switching platform to another configuration, for example by turning a memory to retention state, grows with the number of scenarios, because more scenarios result in more frequent switching. In addition, the runtime manager becomes more complex with an increasing number of scenarios.

Fig. 2. Scenario generation based on profiling information and memory models.

As shown in Fig. 2, using the information available after application profiling, situations with similar platform needs will be organised in a common scenario. This is known as clustering of RTSs [10]. This is a rational choice, because two instances with similar platform needs have similar memory energy consumption. The energy gain of having two dedicated scenarios is small and normally outweighed by switching costs if organized in different scenarios. In this example, clustering results in three different scenario areas (Fig. 2) and the execution scenario sequence is 1(lower), 2(middle), 3(top), 2, 3 and 1. Also, the frequency of occurrence of each instance is an important factor in scenario generation. Instances with higher frequencies normally have their dedicated scenario, to allow higher optimization, while less common instances can be clustered together in a less optimal scenario.

In order to generate scenarios that can be implemented in realistic memory organisations, a detailed library with memory components is needed. The library should contain a variety of memory models, including different technologies and sizes, and is used to calculate scenario costs. In this work the component library is based on memory models published in [20].

The design-time scenario prediction phase consists of determination of the variables that define the active scenario. This can be achieved by careful study of the application code, combined with the application's data input. In our case the grey-box model reveals only the code parts that will influence memory usage, so that variables deciding memory space changes can be identified. An example of this is a non static variable that influences the number of iterations for a loop that performs one memory allocation at each iteration. Moreover, the designer should look for a correlation between input values and the corresponding cost. This information will be useful in the following steps of the methodology [2].

3.4 Run-time Identification, Detection, and Switching

After profiling and scenario generation at design-time, all the necessary information needed for run-time management is available. The run-time manager monitors the current values of prediction variables selected in the previous design time step. Based on this, prediction of the next active scenario is performed [2].

Switching decisions will be taken at run-time by the run-time manager. The switching phase consists of all platform configuration decisions that can be made at run-time, e.g., frequency/voltage scaling, turning on/off a memory unit, and remapping of data on memory units. Switching takes place when the switching cost is lower than the energy gains achieved by switching. In more detail, the run-time manager compares the memory energy consumption of executing the next task in the current active scenario with the energy consumption of execution with the optimal scenario. If the difference is greater than the switching cost, then

Fig. 3. Target platform with focus on memory organisation.

scenario switching is performed [2]. Switching costs are defined by the platform and include all memory energy penalties for run-time reconfigurations of the platform, e.g., extra energy needed to change state of a memory unit.

4 Target Platform

Selection of target platform is an important aspect of the memory-aware system scenario methodology. The key feature needed in the platform architecture is the ability to realize different scenarios generated by the methodology. Execution of different scenarios then leads to different energy costs, as each configuration of the platform results in a specific memory energy consumption. In the conventional system scenario methodology several platform reconfiguration options have been studied [10] with dynamic voltage/frequency scaling (DVFS) most used [21]. By using DVFS techniques, one can change the frequency of the processing element and its supply voltage and energy consumption accordingly. For a memory aware scenario methodology there are multiple ways that memory reconfiguration can be performed. E.g., data can be mapped to different memory units according to size and access frequency, memory units can be turned on or off, or DVFS can be used to allow exactly the access frequency currently needed.

In this paper we have selected a relatively conventional reconfigurable architecture template that is suitable for implementing system scenarios as shown in Fig. 3. This dynamic memory organisation is based on memory prototypes presented in [20], and consists of two software controlled SRAM scratchpad memories, L1 and L1', and a processing element with its registers. For simplicity it is assumed that all data needed during execution is available in the L1 scratchpad memory without any time penalties even when large background memories are used. This assumption is reasonable for the kind of applications that are generally executed in embedded systems and can, e.g., be achieved through prefetching. The methodology is in general not restricted to this assumption, however, and can handle more complex hierarchical memory architectures, also including regular caches. Registers are used to save currently used elements, and between the registers and the L1 scratchpad a much smaller L1' scratchpad is introduced. This is a clustered memory that consists of four memory banks that support three different states (on, off and retention modes) [20].

The memory energy consumption for every access in the clustered scratchpad memory is calculated based on the current situation of the platform as shown in Fig. 3. The leakage energy estimation is also provided in [20] and is included in our study. In Tab. 1 the leakage and wake up costs for all different operation modes are presented. The leakage in L1' is small compared to L1 and constant, thus the effect in the results is minimal.

Table 1. Energy costs per second and access for a reference memory bank (size: 120 bytes, technology: 90nm) in clustered L1' memory

Modes	Leakage [J]	Wake up [J]	Access [J]
On	$351,86 \times 10^{-6}$	-	-
Off	$4,73 \times 10^{-6}$	$2,77 \times 10^{-12}$	-
Retention	$97,33 \times 10^{-6}$	$2,2 \times 10^{-12}$	-
DFF synch	-	-	0.227×10^{-6}
DFF asynch	-	-	2.18×10^{-6}
Nand2	-	-	0.334×10^{-6}

The memory energy consumption per read access is given by Equation 1, which is provided by the memory models in [20].

$$Read\,energy = (size\,of\,bank) \times (DFF\,sync) + \\ + \frac{bank\,lines}{log2} \times DFF\,async + 4 \times size \times Nand2 \quad (1)$$

The cost of accessing the clustered scratchpad organisation is a function of the overall size of the cluster and the size of the specific bank being accessed. In addition, there is a contribution in energy consumption per access added by the flip-flops and the gates needed for the correct operation of the L1' memory organisation.

5 Application Benchmarks

The extended methodology will be tested on two representative real life applications that differentiate significantly from each other and cover different domains of applications. The first one is a computational intensive application, in which the code is dominated by loops with dynamic bounds determined by results calculated inside the loop. The second has a more static execution path, but its memory footprint is dynamically defined by the input data. Ideal applications, that can most benefit from memory-aware system scenario methodology, are applications that have dynamic behaviour in memory organisation utilization during their execution. That required dynamism could be produced by several code characteristics, covering a wide range of potential application domains for the proposed methodology.

5.1 Epileptic Seizure Predictor

An epileptic seizure predictor algorithm developed at Arizona State University (ASU) [22] is chosen as a benchmark, along with a database with measurements performed on real patients, also provided by ASU. Up to now the algorithm has only been used in clinical environments using PCs, although it would be very helpful for patients as an embedded device, as part of outpatient care. Dynamic

Fig. 4. Memory access pattern of epilepsy predictor. Profiling of data reuse size for 3 samples from a given ([22]) database. The number of elements accessed is input dependent. Only those of the 16K elements accessed multiple times should be saved in L1'. The rest are accessed from L1.

input data dependent behaviour is identified in the algorithm and memory traces for execution of three different input samples are presented in Fig. 4. For each data sample a loop is iterated 168 times. Depending on the input data sample, different memory elements are accessed in each iteration. Note that even though the element accessing for all three samples are drawn together in the figure, only one set of memory elements are accessed for each data sample and only one data sample is processed at a time. Because of the nature of the EEG signal used for the epileptic seizure prediction the benchmark has variables with a very wide range of potential values. The memory usage is heavily influenced by these input values. For demonstration purposes our study focuses on the parts of the prediction algorithm that is most dynamic. In [11] the algorithm is split in thread nodes and dynamic behaviour affecting execution time is identified. The data reuse size for each sample in Fig. 4 contains all the elements with data reuse factor greater than 1, i.e., are read more than once. They should be saved in L1'. For example, elements in index range from roughly 3300 to 7200 for sample 1 form an approximately 3900×(element size bytes) L1' memory size requirement. These elements are read for the first time between loop iteration 0 and 50 and are also re-read later during the sample's lifetime between loop iteration 120 and 170. In Fig. 4 these memory elements are included in the red rectangle. The exact L1' sizes needed for processing of sample 1, 2, and 3 are 3899, 2646, and 3780, respectively.

Based on profiling results, scenario generation can be performed. Two possible clustering solutions are used in this paper, both of them consisting of five scenarios. In the first clustering, the range of memory indexes is split in equally sized partitions. In the second clustering, the range is split based on occurrence frequency, so that each scenario has almost equal number of occurrences. The reconfigurable memory platform is instantiated accordingly in our target template outlined in Section 4. It contains four equal memory banks of size 975 in the first case and four banks of increasing size (195, 585, 1170, and 1950) in the second case, since smaller sizes are used more often.

5.2 Viterbi Algorithm Encoder

The Viterbi algorithm [23] is widely used, both in the industry and academia, as an encoding/decoding algorithm for convolutional codes. As part of the encoding of the transmitted signal, redundant bits are added, which are later used by the decoder for correction of transmission errors. The output of an encoder is a function of current input bit(s) and K earlier inputs, where K is the constraint length of the algorithm. Profiling shows that the constraint length dominates the memory cost. In Tab. 2 [24] constraint length values to achieve 10^{-5} BER

Table 2. Constraint length for SNR levels on the channel (BER $\leqslant 10^{-5}$)

SNR (dB)	K	L1' size (B)	SNR (dB)	K	L1' size (B)
6,5 - 6,1	5	40	3,9 - 3,1	9	72
6,1 - 5,5	6	48	3,1 - 2,8	12	96
5,5 - 3,9	8	64	2,8 - 2,5	14	112

Fig. 5. Memory-aware scenario gains - Epileptic seizure predictor

are presented for different noise levels in the channel. Due to its limited range of values, the constraint length is classified as a control variable. The scenario generation is based on profiling and aims at achieving a consistent bit error rate with the minimum memory usage. If SNR decreases, another memory bank is activated while reduction in channel noise leads to the opposite effect. The L1' sizes can be found in Tab. 2. Clustering has been performed for K-values 5 and 6, with a resulting L1' size of 48B, and between K-values 8 and 9, with a resulting L1' size of 72B

6 Results

The memory aware system scenario methodology is applied to both of our benchmark applications to study its effectiveness. Memory energy consumption is calculated based on [20] and is the sum of (energy per access) $\times$ (number of accesses) and energy costs for all transitions between memory modes. The energy for each access is defined by the type and the size of the accessed memory. Based on profiling and scenario generation results, four different memory organizations are compared for the epileptic seizure predictor. The first one is a scenario strategy without memory awareness and use a single memory bank large enough to satisfy the most demanding sample. That approach is the worst case for the memory size and statically allocates the highest value of the memory space demanded during the lifetime of the application, i.e., 3900. However, the number of accesses to the memory will be determined by each sample and no worst case assumption is made for number of accesses.

The second approach assumes that L1' has four banks of the same size (975) that can be turned on and off. The third one assumes an L1' with four banks with different sizes (195, 585, 1170, and 1950), in order to better exploit RTSs with small memory footprints. Finally, a theoretical lower bound assumes an unlimited number of memory banks in all sizes to optimally exploit every situation. Comparison results are shown in Fig. 5 and memory energy gains up to 40% are achieved with the scenario methodology for dynamic input samples. The high quality of our results is substantiated by the fact that our energy consumption is very close to the theoretical lower bound, even though the latter is impossible due to limited area in embedded devices.

Fig. 6. Memory-aware scenario gains - Viterbi encoder

Even higher gains are found for the Viterbi encoder (Fig. 6) compared to a worst-case assumption of the constraint length being equal to 14. That value of length is used to achieve acceptable BER over very noisy channels. Using the extended system scenario methodology, L1' is split into four smaller banks that can be turned off when the noise of the channel is reduced. Again, the theoretical lower bound assumes a memory organisation with bank sizes optimized for each SNR level. Its hardware implementation would need 24 memory banks instead of 4 in our case, which leads to an unacceptable overhead. The proposed reconfigurable memory organisation can lead to more than 70% reduction in memory energy consumption in situations with low noise, compared to a static architecture that is tuned to handle SNR levels down to 2.5 dB.

7 Conclusion

The scope of this work is to extend the existing system scenario techniques to better take memory into account. The memory-aware system scenario methodology has been described and tested on two real applications from bioengineering and wireless communications domains. Results justify the effectiveness of the methodology in reduction of memory energy consumption, which is of great importance in embedded devices. Since memory size requirements are still met in all situations, performance is not reduced. The memory-aware system scenario methodology is suited for applications that experience dynamic behaviour with respect to memory organisation utilization during their execution.

References

1. N. R. Miniskar, "System scenario based resource management of processing elements on mpsoc," Ph.D. dissertation, Katholieke Universiteit Leuven, 2012.
2. Z. Ma *et al.*, *Systematic Methodology for Real-Time Cost-Effective Mapping of Dynamic Concurrent Task-Based Systems on Heterogenous Platforms*, 1st ed. Springer Publishing Company, Incorporated, 2007.
3. R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 9, pp. 1277 –1284, sep 1996.
4. E. Cheung, H. Hsieh, and F. Balarin, "Memory subsystem simulation in software tlm/t models," in *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, jan. 2009, pp. 811 –816.
5. T. Simunic, L. Benini, and G. De Micheli, "Cycle-accurate simulation of energy consumption in embedded systems," in *Design Automation Conference, 1999. Proceedings. 36th*, 1999, pp. 867 –872.
6. L. Benini, A. Macii, and M. Poncino, "A recursive algorithm for low-power memory partitioning," in *Low Power Electronics and Design, 2000. ISLPED '00. Proceedings of the 2000 International Symposium on*, 2000, pp. 78 – 83.

7. L. Benini *et al.*, "Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation," *Design Test of Computers, IEEE*, vol. 17, no. 2, pp. 74 –85, apr-jun 2000.
8. A. Macii, L. Benini, and M. Poncino, *Memory Design Techniques for Low-Energy Embedded Systems*. Kluwer Academic Publishers, 2002.
9. P. R. Panda *et al.*, "Data and memory optimization techniques for embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 6, no. 2, pp. 149–206, Apr. 2001.
10. S. V. Gheorghita, *et al.*, "System-scenario-based design of dynamic embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, pp. 3:1–3:45, Jan. 2009.
11. E. Hammari, F. Catthoor, J. Huisken, and P. G. Kjeldsberg, "Application of medium-grain multiprocessor mapping methodology to epileptic seizure predictor," in *NORCHIP, 2010*, nov. 2010, pp. 1 –6.
12. M. Palkovic, F. Catthoor, and H. Corporaal, "Dealing with variable trip count loops in system level exploration." in *ODES: 4th Workshop on Optimizations for DSP and Embedded Systems*, 2006.
13. M. Palkovic *et al.*, "Systematic preprocessing of data dependent constructs for embedded systems," *Journal of Low Power Electronics, Volume 2, Number 1*, 2006.
14. M. Palkovic, H. Corporaal, and F. Catthoor, "Heuristics for scenario creation to enable general loop transformations," in *System-on-Chip, 2007 International Symposium on*, nov. 2007, pp. 1 –4.
15. P. Garcia, K. Compton, M. Schulte, E. Blem, and W. Fu, "An overview of reconfigurable hardware in embedded systems," *EURASIP J. Embedded Syst.*, vol. 2006, no. 1, pp. 13–13, Jan. 2006.
16. S. Himpe, G. Deconinck, F. Catthoor, and J. van Meerbergen, "Mtg* and grey-box: modelling dynamic multimedia applications with concurrency and non-determinism," in *System Specification and Design Languages: Best of FDL02*, 2002.
17. F. Catthoor, S. Wuytack, G. de Greef, F. Banica, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Springer, 1998.
18. D. Patterson and J. Hennessy, *Exploiting Memory Hierarchy in Computer Organization and Design the HW/SW Intelface*. Morgan Kaufmann, 1994.
19. N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, 2007.
20. A. Artes *et al.*, "Run-time self-tuning banked loop buffer architecture for power optimization of dynamic workload applications," in *VLSI and System-on-Chip (VLSI-SoC), 2011 IEEE/IFIP 19th International Conference on*, oct. 2011, pp. 136 –141.
21. A. Portero *et al.*, "Dynamic voltage scaling for power efficient mpeg4-sp implementation," in *Application-specific Systems, Architectures and Processors, 2006. ASAP '06. International Conference on*, sept. 2006, pp. 257 –260.
22. L. Iasemidis *et al.*, "Long-term prospective on-line real-time seizure prediction," *Clinical Neurophysiology*, vol. 116, no. 3, pp. 532–544, 2005.
23. A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *Information Theory, IEEE Transactions on*, vol. 13, no. 2, pp. 260 –269, april 1967.
24. S. Swaminathan *et al.*, "A dynamically reconfigurable adaptive viterbi decoder," in *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-*

programmable gate arrays, ser. FPGA '02. New York, NY, USA: ACM, 2002, pp. 227–236.