Automated Generation of Basic Custom Sensor-Based Embedded Computing Systems Guided by End-User Optimization Criteria

Susan Lysecky[1] and Frank Vahid[2]

[1]Department of Electrical and Computer Engineering
University of Arizona
Tucson, AZ 85721
slysecky@ece.arizona.edu

[2]Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA 92521
vahid@cs.ucr.edu
Also with the Center for Embedded Computer Systems at UC Irvine

Abstract. We describe a set of fixed-function and programmable blocks, *eBlocks*, previously developed to provide non-programming, non-electronics experts the ability to construct and customize basic embedded computing systems. We present a novel and powerful tool that, combined with these building blocks, enables end-users to automatically generate an optimized physical implementation derived from a virtual system function description. Furthermore, the tool allows the end-user to specify optimization criteria and constraint libraries that guide the tool in generating a suitable physical implementation, without requiring the end-user to have prior programming or electronics experience. We summarize experiments illustrating the ability of the tool to generate physical implementations corresponding to various end-user defined goals. The tool enables end-users having little or no electronics or programming experience to build useful customized basic sensor-based computing systems from existing low-cost building blocks.

1 Introduction

The cost, size, and power consumption of low-end microprocessors in the past decade has dropped tremendously as silicon technology continues to follow Moore's Law. For example, an 8-bit microprocessor chip may cost less than $1, occupy just a few millimeters, and consume just microwatts of power. Such reductions enable integration of microprocessors into domains previously unthinkable, such as RFID tags, ingestible pills, and pen tips.

Meanwhile, a problem in the design of basic sensor-based embedded computing systems is that end-users cannot setup basic custom embedded systems without the assistance of engineers. An end-user is an individual developing a sensor-based computing system who likely does not have programming or electronics expertise,

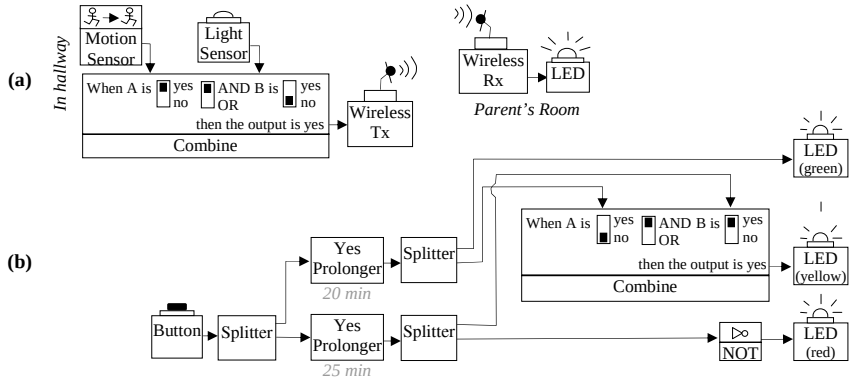

Figure 1. Example sensor based applications built with fixed-function blocks, (a) Sleepwalking Detector, (b) Presentation Timer.

such as a homeowner, teacher, scientist, etc. For example, a homeowner may wish to setup a custom system to indicate that a garage door is open at night, that a child is sleepwalking, or that an ageing parent has yet to get out of bed late in the morning. A scientist may wish to setup an experiment that activates a video camera when an animal approaches a feeding hole, or activates a fan when a temperature exceeds a threshold. Countless other examples exist. Despite the fact that such systems could be built from computing and sensing components whose total cost is only a few dollars, end-users cannot build such systems without knowledge of electronics and programming. Just connecting a button to an LED (light-emitting diode) would require knowledge of voltages, grounding principles, power supplies, etc. Making such a connection wireless requires further knowledge of communications, microcontroller programming, wireless devices, etc. Hiring engineers to build the system immediately exceeds reasonable costs. Off-the-shelf solutions for specific applications are hard to find, costly due to low volumes, and difficult to customize.

Our previous work addressed this problem through development of basic blocks, called *eBlocks*, that enable end-users with no electronics or programming experience to define customized sensor-based systems merely by connecting blocks and performing minor configuration of those blocks [5]. The blocks incorporate small inexpensive microprocessors into previously passive devices like buttons, motion sensors, and beepers. Each device has a fixed function, and can be easily connected to other devices merely by snapping together standard plugs, with the devices communicating using predefined basic networking protocols. Figure 1 illustrates several applications built using eBlocks. A sleepwalking detection application is shown in Figure 1(a), which consists of motion and light sensor outputs combined using a logic block (configured to compute motion and no light), whose output connects to a wireless transmit block. The wireless transmit block is matched with a wireless receiver block (through setting of switches to identical positions), which ultimately activates a beeper block when motion at night is detected. Figure 1(b) illustrates a second example, a presentation timer, which turns on a green light for 20 minutes, followed by a yellow light for 5 minutes, and lastly a red light indicating time has expired. The design splits a button press to two prolonger blocks, one that

prolongs the button press for 20 minutes, another for 25 minutes. Additional logic turns on the appropriate lights depending on whether both, one, or neither prolonger block is outputting a true value.

eBlocks represent one of several new research approaches that utilize physical ("tangible") objects to enable end-users to program electronic devices [12]. Other examples include Media Cubes [1], Electronic Blocks [20], and Tangible Programming Bricks [16]. Commercial X10-based devices [18] communicate through household power lines and are complementary to our approach.

This paper describes our recent efforts to develop computer-based tools that an end-user could use to optimize an eBlock system or to map a virtual eBlock system to a limited set of physical blocks. While the work in this paper describes a tool to help end-users to *build* eBlock systems, related work that we have done [15] describes a tool to help end-users to *tune* low-level eBlock parameters, such as microprocessor clock frequency and communication baud rate, in order to achieve goals like maximizing battery lifetime and/or minimizing system latency.

2 eBlocks Overview

The key idea of eBlocks is to enable end-users to build useful customized sensor-based systems merely by connecting blocks, like buttons, motion sensors, logic, beepers, etc. eBlocks' key feature is that they encapsulate previously passive components by an ultra-lightweight compute wrapper. The following sections briefly describe two types of eBlocks, fixed function blocks and programmable blocks. A section also describes the eBlocks simulator, a multifaceted graphical environment that can simulate system functionality, configure programmable blocks, and provide the interface for the technology mapping and optimization tool introduced in this paper. Further details on eBlocks are discussed in [6].

2.1 Fixed-Function Blocks

Fixed-function eBlocks have a specific predefined function. Two types of fixed-function eBlocks are Boolean and integer. Boolean blocks send "yes" or "no" packets, while integer blocks send integer packets. While this paper focuses on Boolean blocks, the methods generally apply to integer blocks, and our future work will address such application. Four categories of Boolean blocks exist: sensor, compute, communication, and output blocks.

Sensor blocks sense events, such as motion, light, sound, button presses, or temperature. When a sensor detects the presence of an event (i.e. a light sensor detects light), the sensor generates a yes output, and otherwise generates a no output.

Compute blocks perform logic or state computations on inputs and generate new outputs. A 2-input "Combine" block (a.k.a. "Logic") computes a 2-input logic function configured by the end-user (e.g., AB, or A'+B). A 3-input Combine block is also available. An inverter block inverts a yes input to no output, or a no input to yes output. A "Yes Prolonger" block prolongs a yes input over the block's output for an

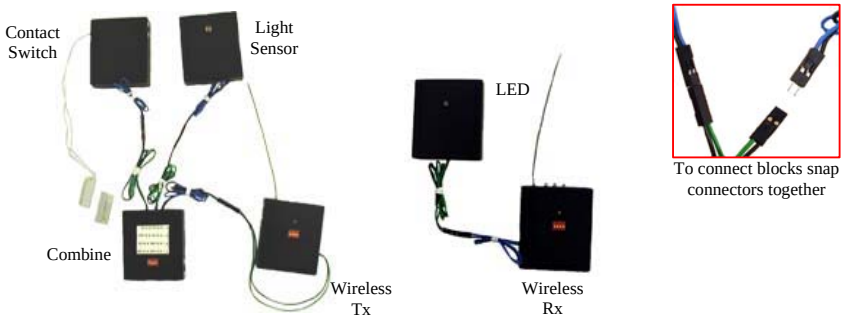

Figure 2. Garage Door Open At Night Dector is built by snapping various fixed function eBlocks together.

end-user-configured duration. A "Toggle" block switches between yes and no outputs on successive yes inputs. A "Pulse Generator" block generates yes and no output pulses for an end-user-configured duration. A "Once-Yes Stays-Yes" (a.k.a. "Tripper") block trips to a yes output state when the main input receives a yes, and stays in that state until a yes appears on a reset input.

Communication blocks include wireless transmit and wireless receive blocks, which must be configured to implement a point-to-point channel by setting the corresponding switches on each block to the same channel value. A splitter block splits a single input into multiple identical outputs.

Output blocks beep, turn on LEDs, control electric relays, or provide data to a PC for logging or other processing. A yes input activates output blocks. For example, a beeper block beeps when its input is yes, and is silent when its input is no. Figure 2 shows our initial physical prototype versions of eBlocks. Each physical block contains a PIC microcontroller for local computation and inter-block communication. The connections among blocks (along with any configurations of each block) define a system's functionality. A unidirectional, packet-based protocol provides the basis for block communication. Each block includes hardware specific to the block's task (e.g., sensors, resistors, voltage regulators, etc.). An end-user connects blocks using wired connectors or can replace a wire by a wireless connection by utilizing wireless transmit and receiver blocks. We have built over 100 prototype physical blocks, successfully used in controlled experiments by over 500 people of various skills levels, mostly end-users with no programming background [5].

2.2 eBlock Simulator

The eBlock simulator, shown in Figure 3, is a Java-based graphical user interface (GUI) for eBlock system entry and simulation and is available online at [6]. End-users can connect, test, and optimize various eBlock systems before interacting with physical blocks. End-users drag a block from a catalog on the right edge of the simulator to the workspace on the left and connect the blocks by drawing lines between the blocks' input and outputs. The user can choose between a "simple mode"

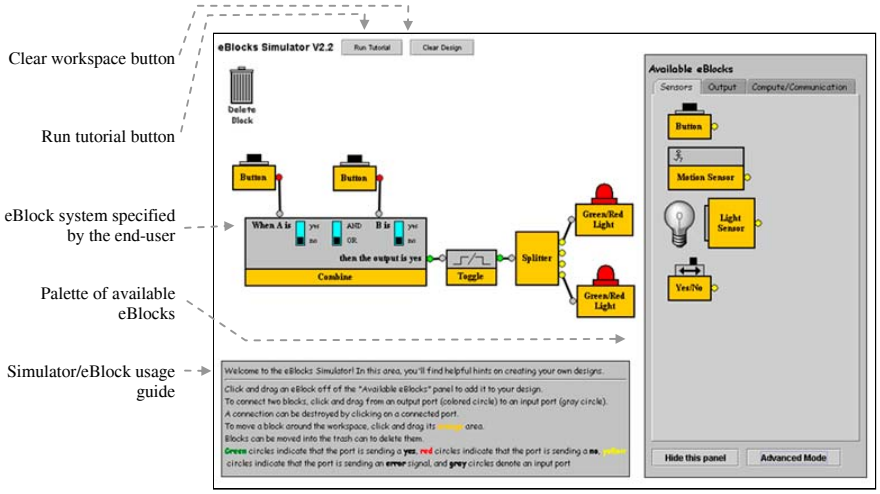

Figure 3. Capture and synthesis tool illustrating the cafeteria food alert system.

in which commonly-blocks appear in the catalog, and an "advanced mode" containing more blocks. eBlocks that sense or interact with their environment include accompanying visual representation to simulate the corresponding environment. For example, a "day/night" icon accompanies the light sensor. End-users can alternate the icon between day and night by clicking on the icon, causing the light sensor's output to change accordingly. The gray text box situated in the bottom-left of the simulator displays context-sensitive help such as a block's description and interface when the mouse cursor hovers over the corresponding block.

2.3 Programmable Blocks

In contrast to fixed-function blocks, a programmable block can be programmed to implement arbitrary behavior. Our current programmable block has two inputs and two outputs, as shown in Figure 4. An expert user could write C code that describes the block's behavior, and our tools would then combine that with the eBlock protocol code into a binary, which the user could then download using our serial cable interface.

However, most end-users will not have C-coding expertise. We thus provide a tool for automatically converting internal (non-sensor, non-output blocks) fixed-function blocks into a smaller network of programmable blocks that preserves the system's functionality, automatically generating code for each programmable block. In this context, a programmable block is a means for slightly more advanced users to reduce the block count and hence cost of their systems.

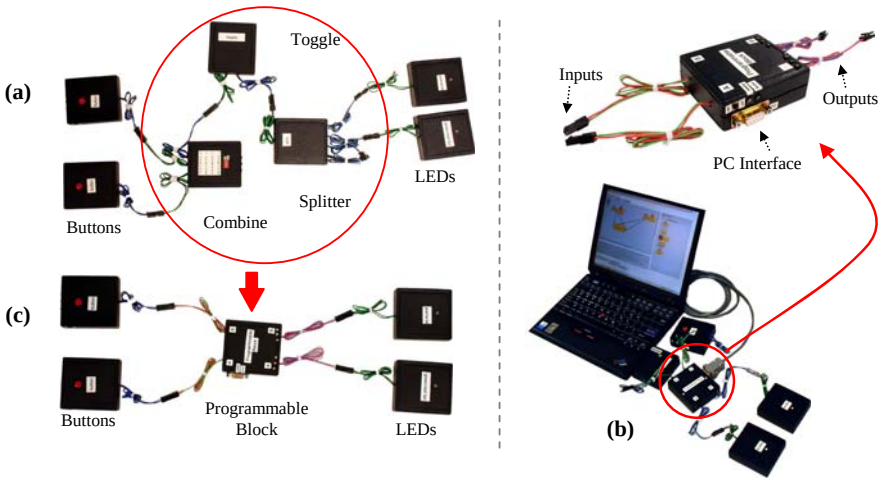

Figure 4. Programmable blocks: (a) An original system using fixed-function blocks, (b) programming a programmable block to replace the inner fixed-function blocks, (c) the new system using the programmable block.

3 Technology Mapping

3.1 Problem Description

A problem with using physical fixed-function blocks is that an end-user may only have a subset of possible blocks readily available, and/or may have limited numbers of particular blocks. For example, an end-user may have three types of blocks available – 2-input logic, tripper, and prolonger blocks – of which many instances of each exist. Defining the desired application behavior using just those blocks would be a significant challenge even for an expert end-user. Instead, we provide for the end-user a way to capture the desired behavior in the graphical simulator using any combination of standard fixed-function blocks. The end-user also lists the types and numbers of available physical fixed-function blocks, forming what is essentially a constrained block library. We then define an automation tool that creates a new block network with the same functionality as the desired network, but only using blocks from the constrained library. Sensor and output blocks have specialized circuitry (e.g., light sensors, LEDs, beepers), so the physical counterparts for those blocks must be available – i.e., we cannot build a light sensor out of motion sensors. The mapping problem thus only involves *inner* blocks, namely compute and communication blocks.

The above-described problem is essentially a *technology-mapping* problem, common in chip design, with some differences from traditional problems. Technology mapping is a central part of the chip design process. Chip designers describe a desired circuit's behavior using easy-to-work-with components, such AND, OR, and NOT logic gates, with any number of inputs on each gate. However, a chip's underlying

technology may support only 2-input and 3-input NAND gates, requiring that generic AND/OR/NOT circuits be mapped to a circuit consisting only of such NAND gates. Modern technologies consist of far more complex mappings of generic circuits to technology-specific components. Efficient technology mapping has been intensely researched for decades [2,3,4,8,9]. Our technology-mapping problem has some differences from previous technology-mapping problems. In the domain of field programmable gate arrays (FPGAs), technology mapping translates a digital circuit to a physical implementation on lookup tables (LUTs) [7]. A LUT is capable of specifying any logic function with a given number of inputs (defined by the size of the physical LUTs within the FPGA). Because LUTs are general programmable structures, the mapping methods correspond more to the problem of converting to programmable blocks than to that of fixed-function blocks. In application-specific integrated circuit (ASIC) design, technology-mapping implements a hardware circuit using a library of physical cells with fixed functions [7]. However, the final ASIC implementation can use an essentially unlimited number of any physical cells within the library. In contrast, our problem has a fixed numbers of each block, and furthermore does not necessarily have a balanced set of blocks. Nevertheless, our solution approach borrows from existing ASIC techniques.

3.2 Transformation Rule Base

A straightforward but non-optimal ASIC technology mapping method converts every technology independent circuit element into a technology-dependent universal gate element. In circuits, a NAND gate or a NOR gate represents a universal gate. In our problem, we found that we could implement nearly every block using some network of 2-input logic and splitter blocks. Defining a universal mapping heuristic that would replace each unmapped block in a network by an equivalent network consisting of the universal blocks is one possible method to perform technology mapping.

A better optimizing ASIC technology-mapping method involves graph-covering methods [13]. A library is built of mappings from technology independent sub-circuits to technology dependent sub-circuits, and then directed acyclic graph covering methods cover the unmapped circuit. The methods are built on similar graph methods used for instruction coverage generation in compilers. While such methods could be applied to our problem, we found that the state-based functions associated with our fixed-function blocks might introduce significant complexity into the graph cover heuristics. In fact, such traditional methods typically focus on the combinational part of the circuit, whereas state-based (sequential) blocks are a key part of our problem.

Another ASIC technology mapping method involves rule-based technology mapping [10,11]. Those techniques perform local optimizations on a circuit based on a set of transformation rules. We used this method as the basis for our first solution to the problem, which we call the transformation rule method. We developed a transformation rule base as follows. For each standard fixed-function block, we manually built alternative implementations of that block (the source block) using other various subsets of standard fixed-function blocks (target blocks). For example, for a 2-input logic block, we defined a transformation for implementing that block using a 3-input logic block, as shown in Figure 5(a) (*Config*. shows the truth table

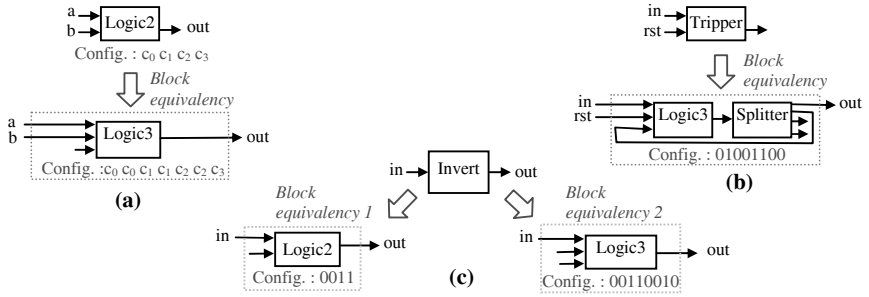

Figure 5. Sample block equivalencies used in the technology mapping equivalncy library.

entries for the block). Figure 5(b) shows a transformation of a tripper block into an equivalent set of blocks involving a logic block and a splitter block, with the required logic configuration shown. Figure 5(c) shows multiple transformation rules for a inverter block. The invert block can be replaced utilizing either a 2-input logic block or a 3-input logic block, configured to implement the invert on the first input.

We point out that we could have treated logic blocks using logic synthesis methods, wherein we would convert every sub-network of logic blocks (2- and 3-input logic blocks and invert blocks) into a Boolean expression, optimize the expression, and then map the expression into logic blocks in the library using traditional circuit technology mapping. However, as this is a first work, we preferred a transformation rule approach for consistency with the other blocks, resulting in a simpler tool but a less optimized mapped network. Nevertheless, incorporating logic synthesis methods is an area of future improvement.

4 Optimization

4.1 Problem Description

Given a network of fixed-function blocks, there may exist more blocks than necessary, arising from two situations. First, an end-user may have created a network of fixed-function blocks that is easy to comprehend, but has more blocks than necessary. Alternatively, technology mapping may have inserted two adjacent sub-networks with perimeter blocks that could be merged into fewer blocks. We thus developed a method to reduce block count while preserving network behavior.

We considered different methods for reducing blocks. A model-based method would utilize a formal understanding of the underlying finite-state machine (FSM) (or combinational) behavior of each block. This method would compose the FSMs into a single network-level FSM, eliminate equivalent and redundant states, and remap the reduced FSM to physical blocks. This approach appeared overly complex and possessed the problem that the reduced FSM might not be mappable to existing physical blocks. Another method builds on peephole optimization, an optimization method commonly found in compilers. This method inspects a local area of code to identify and modify inefficient code [17]. We can similarly inspect sections of the

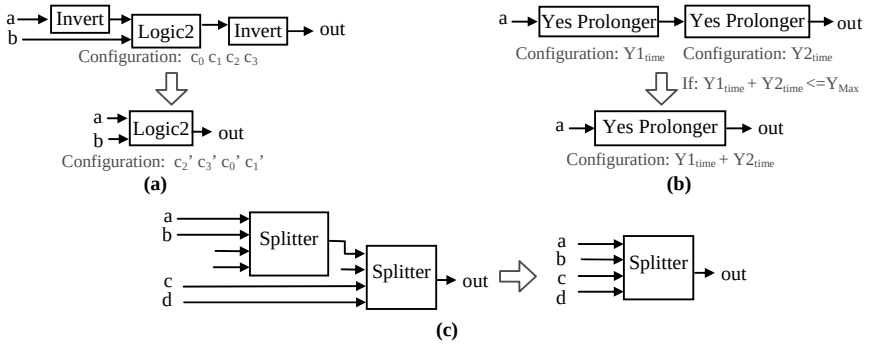

Figure 6. Sample peephole optimization used in the optimization library.

block network to optimize inefficient or redundant sections. The peephole optimization method enables us to preserve the pre-defined block structure because we are operating on the block level and eliminates the need for additional mapping.

4.2 Peephole Optimizations

We analyzed a variety of networks and identified commonly occurring inefficient block combinations. We added optimization templates to the library that reflect these inefficient block combinations, along with a corresponding optimized block network. The optimizer traverses the network specification searching for subsystems matching any of the corresponding optimization templates and replaces the inefficient block combination by the optimized block network defined by the template.

Figure 6 illustrates several optimization templates defined within the optimization library. Figure 6(a) illustrates inverters located at the input or output of logic blocks that an end-user could have merged into the logic block. The optimizer eliminates the inverters and updates the logic block configuration accordingly. The optimization shown in Figure 6(b) merges chained prolonger blocks into a minimum number of prolonger blocks. If the combined yes time of the chained prolonger blocks is less than the maximum yes time of a single prolonger block, the optimizer can merge the chained prolonger blocks into a single prolonger block. If the combined yes time of the chained prolonger blocks exceeds the maximum yes time of a single prolonger block, then the minimum number of prolonger block are used. Figure 6(c) analyzes the number of unused inputs on chained splitters and attempts to combine splitters. Each peephole optimization is treated independent of others peephole optimizations as well as independent of the technology mapping transformations.

5 Programmable Block Operations

Technology mapping transformation rules and peephole optimizations discussed in previous sections pertain to fixed function blocks. Inclusion of programmable blocks

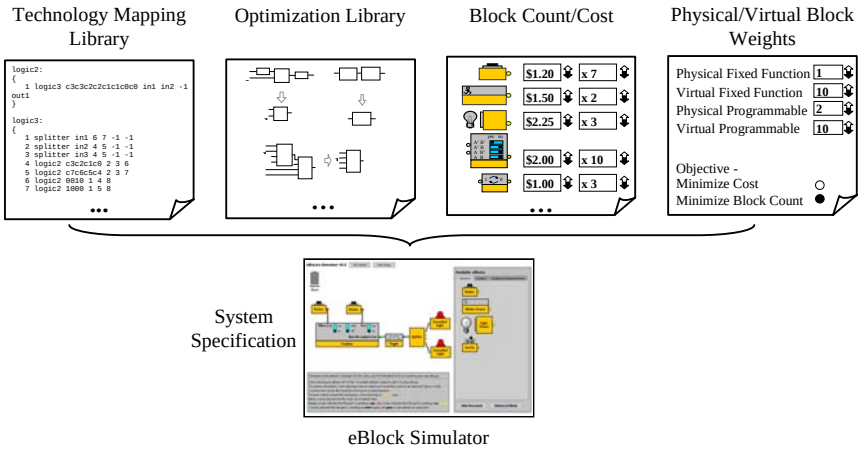

Figure 7. Technology mapping and optimization environment.

presents further opportunity for technology mapping and optimization. For example, if a fixed function block does not exist in the physical library, a programmable block can by configured to replace the missing fixed function block. Furthermore, multiple fixed-function blocks can be replaced by a single programmable block to reduce block count and/or cost of the system.

Several options exist to deal with the existence of programmable blocks. One option is to develop a separate partitioning algorithm, utilized in a secondary stage, which aims to assign multiple fixed-function blocks to programmable blocks. A simpler method is to define low-level technology mapping transformation rules and peephole optimizations specific to programmable blocks, and to incorporate those rules and optimizations into the main technology mapping and optimization heuristic (discussed in Section 6). The second method follows closely what we have done with fixed-function blocks, thus we defined several programmable block operations and incorporated them into the appropriate libraries.

6 Technology Mapping and Optimization Methodology

Figure 7 illustrates the overall technology mapping and optimization design methodology intended to aid end-users in generating an optimized physical sensor-based system based on end-user defined criteria. Two parties are responsible for the input specification, the node designer and the end user. The node designer is an expert who has an understanding of the underlying details of the various eBlocks and provides the pertinent block information prior to the release of the mapping and optimization tool. The end-user may have no expertise in programming or electronics but wants to construct a customized sensor-based system. The end-user provides input specific to their situation and the application being created.

6.1 Node Designer Input

The node designer defines the technology mapping transformations discussed in Section 3 by creating a text-based technology-mapping library read in by the tool. Furthermore, the node designer defines various peephole optimizations discussed in Section 4 utilized by the tool. The optimization library is currently a C file that contains various functions to perform each of the peephole optimizations, alternatively these optimizations could be defined in a text file as the technology mapping transformation which is then translated by the technology mapping and optimization tool. These files are provided by the node designer and are independent of the various sensor-based systems constructed by the end-user.

6.2 End-User Input

The end-user needs to specify which fixed-function and programmable blocks are physically available, the functionality of the specific application being built, and the optimization goals.

The end-user first defines the "Block Count/Cost" input, i.e., how many of each type of block is physically available and the cost of each eBlock (regardless of whether they are physically available). The input specification can be done in a graphical environment in which an end-user can manually enter a number in a text box next to the graphical depiction of the block of interest or click on up/down arrows until the appropriate value is displayed (shown in Figure 8 under the "Block Count/Cost" heading). The end-user can similarly define block cost.

The end-user's next task is to define the "Physical/Virtual Block Weights" input. We define physical blocks as blocks that are physically available in the end-user's block set; virtual blocks do not exist in the block set, meaning they would have to be purchased to create the physical system. The end-user specifies four weights:

W_{P_FF} = *physical fixed function block weight*
W_{V_FF} = *virtual fixed function block weight*
W_{P_PROG} = *physical programmable block weight*
W_{V_PROG} = *virtual programmable block weight*

These values are used within a cost equation to evaluate whether a given system configuration yields an improvement (further discussed in Section 6.3). By simply assigning various weights, with lower weights indicating preferred block types, an end-user can direct the technology mapping and optimization to use preferred block types when possible. For example, an end-user who is uncomfortable with programmable blocks and wants to only utilize fixed function blocks, whether virtual or physical, can set the blocks weights to $W_{P\_FF} = 1$, $W_{V\_FF} = 1$, $W_{P\_PROG} = 10$, and $W_{V\_PROG} = 10$. Selection of a programmable block by the tool yields ten times the cost of a fixed function block, guiding the tool to favor fixed-function blocks. Alternatively, an end-user who wants to utilize blocks already existing in their physical block set, whether fixed or programmable, can set the block weights to $W_{P\_FF} = 1$, $W_{V\_FF} = 25$, $W_{P\_PROG} = 1$, and $W_{V\_PROG} = 25$. Again, virtual blocks yield higher cost, thus the tool is biased to select physically available blocks before utilizing any

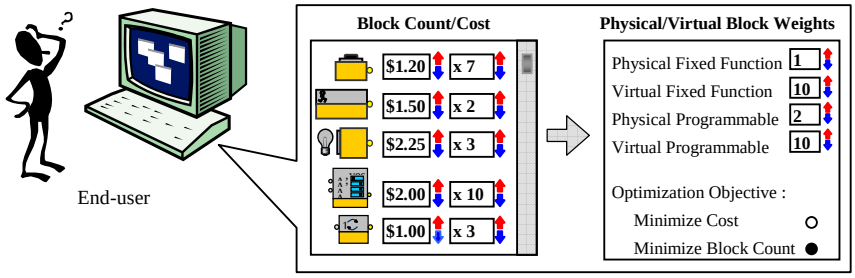

Figure 8. Using a graphical interface, the end-user specifies, (a) the block count/cost library and, (b) the physical/virtual block weights.

virtual blocks. The end-user can adjust the four block weights to reflect a variety of situations and to guide the technology mapping and optimization tool in creating an appropriate physical sensor based system.

Within the "Physical/Virtual Block Weight" input, the end-user must also choose the optimization criteria – either to minimize the number of blocks utilized or to minimize the monetary cost of the resulting system. The end-user selects the optimization goal by selecting the corresponding radio button.

The last task required of the end-user is to create the eBlock system, thus defining the desired system functionality. The end-user creates the eBlock system within the block simulator (Section 2.2) by dragging and connecting blocks in the workspace.

6.3 Design Space Exploration

Once all inputs to the technology mapping and optimization tool are defined, our tool uses simulated annealing to explore the design space and generate the finalized sensor based system. The simulated annealing algorithm [14] is a popular optimization approach modeled after annealing in metallurgy, wherein a material is continuously heated and cooled to increase the material's strength. The algorithm randomly searches the design space by generating random changes, and accepts a change if an objective function value is decreased. Alternatively, a change that increases the objective function value can also be accepted based on a probability linked to a global "temperature" value. Early in the algorithm, changes that increase the objective function cost are more likely to be accepted, to avoid being trapped in a local minimum early on. As the algorithm continues to run, these higher cost changes are less likely to be accepted, thus settling into a minimum cost solution. The rate of decline in which higher cost solutions are accepted is based on a definable cooling schedule. The longer the algorithm runs, the higher a chance of a good solution, thus the key is to define a cooling schedule that balances the solution quality and runtime. We chose simulated annealing due to the heuristic's generality – we can simply define a set of possible changes consisting of the various transformations and optimizations discussed in previous sections, and let the tool search the solution space. While computationally expensive, the power of modern computers coupled with the relatively small sizes of eBlock systems make the use of annealing effective.

Depending on the end-user defined optimization criteria, we use one of two weighted cost functions to determine the system cost. If minimizing block count is the objective, the following cost equation is utilized:

$$\text{block cost} = \left(W_{P\_FF} * \#\ of\ physical\ fixed\ function\ blocks\right) + \left(W_{V\_FF} * \#\ of\ virtual\ fixed\ function\ blocks\right) + \left(W_{P\_PROG} * \#\ of\ physical\ programmable\ blocks\right) + \left(W_{V\_PROG} * \#\ of\ virtual\ programmable\ blocks\right)$$

If minimizing total system cost is the objective, the following cost equation is utilized:

$$\begin{aligned}\text{system cost} = {} & W_{P\_FF} * \sum_i \left(price\ of\ block_i * \#\ of\ physical\ fixed\ fucntion\ block_i\right) + \\ & W_{V\_FF} * \sum_i \left(price\ of\ block_i * \#\ of\ virtual\ fixed\ fucntion\ block_i\right) + \\ & W_{P\_PROG} * \sum_i \left(price\ of\ block_i * \#\ of\ physical\ programmab\ le\ block_i\right) + \\ & W_{V\_PROG} * \sum_i \left(price\ of\ block_i * \#\ of\ virtual\ programmab\ le\ block_i\right)\end{aligned}$$

The end-user, as described in Section 5.2, assigns the various block weights. The technology mapping and optimization tool utilizes simulated annealing and performs random changes consisting of an optimization or a transformation from a randomly chosen single technology mapping transformation rule (Section 3), a peephole optimization (Section 4), or a programmable block operation (Section 5).

7 Technology Mapping and Optimization Results

Utilizing the technology mapping and optimization methodology previously discussed, we now consider several eBlock systems and provide the corresponding physical eBlock system implementation determined by the technology mapping and optimization tool taking into consideration user-specified block availability and preferences. We considered six different scenarios in which end-users have varying types and quantities of physical eBlocks already available, as well as differing preferences as to the types of eBlocks the end-user wants to utilize to build the desired eBlock system. Section 7.1 looks at scenarios where the optimization criterion selected is block size reduction and Section 7.2 looks at scenarios where the optimization criterion selected is system cost reduction.

For each scenario, we considered sixteen eBlock system specifications, ranging from a "Night Light Controller" consisting of two internal blocks to a "Digital Hourglass Timer" consisting of over 50 internal blocks. The "Technology Mapping" library and "Optimization" library input files specified by the node developer are consistent across each scenario considered. The "Block Cost" input contains the monetary cost of each block type, derived from [19], are also consistent across each scenario considered. Each scenario defines the "Block Counts" input specifying the number and type of physical blocks already available to the end-user, i.e., the eBlocks that end-user currently has on hand. Each scenario further defines the "Physical/Virtual Block Weight" input specifying the end-user's preference towards

available physical blocks or virtual blocks the end-user wants to utilize in constructing the final eBlock system implementation.

7.1 Minimizing Block Count

We first consider a scenario, referred to as Scenario 1, in which an end-user may have just stumbled upon eBlocks online and wants to try to build various systems using the eBlock simulator and has no physical eBlocks available. Block Set A, listed in Table 2, reflects that the end-user has no physically available blocks. Furthermore, the Block Set corresponds to the "Block Count" input of the technology mapping and optimization tool. To realize the eBlock system specified within the eBlock simulator, the end-user is willing to purchase fixed function blocks but is weary of utilizing programmable blocks. Thus, the end-user can specify this preference by adjusting the "Physical/Virtual Block Weights" input, setting the fixed function block weights to 10 and programmable block weights to 100, as listed in Table 1. Having no physical eBlocks (as specified by Block Set A) and a desire to utilize only fixed function block (as specified by the block weights listed in Table 1), we then utilized the technology mapping and optimization tool to implement each of the sixteen eBlock systems. Figure 9 illustrates a breakdown of block types for each system, indicating the number of physical fixed function, physical programmable, virtual fixed function, and virtual programmable blocks each eBlock system is composed of. Only a subset of systems is illustrated in Figure 9 due to space limitations. In Scenario 1, programmable blocks are penalized, thus all sixteen final eBlock systems consist solely of fixed function blocks with solutions yielding and average of 12.9 inner blocks.

Block Set	Physical Block Count												
	2-Input Logic	3-Input Logic	Inverter	Toggle	Prolonger	Tripper	Pulse Generator	Wireless Transmitter	Wireless Receiver	Splitter	Prog_2_2	Prog_4_4	Prog_6_6
Block Set A	0	0	0	0	0	0	0	0	0	0	0	0	0
Block Set B	10	2	2	10	2	2	2	10	10	10	0	0	0
Block Set C	10	2	2	10	2	2	2	10	10	10	2	2	2

Table 2. Breakdown of physical block counts for each block set, all input and output blocks are assumed to contain unlimited corresponding physical block counterparts.

	Block Count Assignment	W_{P_FF}	W_{V_FF},	W_{P_PROG}	W_{V_PROG}
Scenario 1 & 7	Block Set A	10	10	100	100
Scenario 2 & 8	Block Set A	10	10	10	10
Scenario 3 & 9	Block Set B	1	10	1	100
Scenario 4 & 10	Block Set B	1	10	1	10
Scenario 5 & 11	Block Set C	1	10	1	100
Scenario 6 & 12	Block Set C	1	10	1	10

Table 1. Breakdown of physical block counts and weights for each sceanrio.

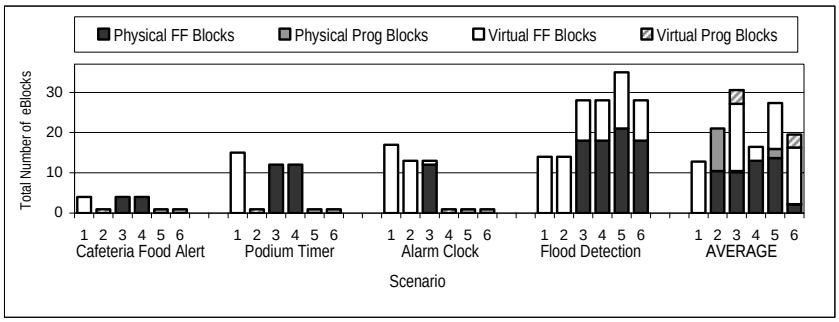

Figure 9. Resulting number of physical fixed function (FF) blocks, physical programmable (Prog) blocks, virtual fixed function blocks, and virtual programmable blocks for several scenarios and systems.

In the second scenario (Scenario 2), the end-user again has no physical blocks available, but the end-user in this scenario is willing to purchase fixed function and programmable blocks. Again, as the end-user has no available blocks, Block Set A is utilized to specify the "Block Count" input. Furthermore, as the end-user does not have a preference towards fixed function or programmable blocks, all block types are equally weighted in the "Physical/Virtual Block Weight" input. Figure 9 illustrates a breakdown of block types for a subset of the sixteen eBlock systems considered. Overall, five solutions can take advantage of programmable blocks, resulting in solutions that on average require 10.5 inner blocks. By allowing programmable blocks, the final eBlocks systems can be implemented using on average 2.4 fewer inner blocks compared to a system composed of solely fixed function blocks (Scenario 1).

In the next two scenarios considered (Scenario 3 and 4), an end-user has access to some physical fixed-function blocks, perhaps having purchased a initial set eBlocks consisting of only fixed function blocks, with the number and type of physical blocks available listed in Block Set B. In addition, the end-user in Scenario 3 is willing to purchase fixed function blocks if needed but is apprehensive to purchase programmable blocks. Block weights are set to so physical blocks have lower weights, virtual fixed-function blocks are weighted slightly higher, and virtual programmable blocks are heavily weighted, as listed in Table 1. In Scenario 4, an end-user is willing to purchase fixed function and programmable blocks. Virtual blocks have slightly higher weights, but the block weights make no distinction between fixed-function and programmable blocks. In Scenarios 3 and 4, a limited number of physical fixed function blocks exist, thus the tool will bias solutions to utilize physically available blocks before choosing virtual blocks as shown in Figure 9. Scenario 3 further penalizes usage of virtual programmable blocks, thus no solutions include virtual programmable blocks. On average, solutions require 16.44 inner blocks but only require end-users to acquire an additional 3.44 inner blocks. While Scenario 3's final inner block count is higher than in Scenarios 1 and 2, existing blocks are utilized minimizing the number of additional blocks required. Scenario 4 yields solutions with an average inner block count of 13.7, of which an

average of 2.3 additional blocks are required. Again, this occurs because the programmable blocks are utilized, enabling reduction of fixed-function blocks.

In Scenarios 5 and 6, an end-user has a slightly larger set of physical fixed function and programmable blocks available as listed in Block Set C. The end-user in Scenario 5 is willing to purchase fixed function blocks, thus the end-user sets W_{V_FF} to 10 while W_{V_PROG} is set to 100. Lastly, in Scenario 6, an end-user is willing to purchase fixed function and programmable blocks. While higher than the physical block weights, both virtual block weights (W_{V_FF} and W_{V_PROG}) are equally weighted. In scenario 5 and 6, a larger physical block library exists, including both fixed function and programmable blocks. Figure 9 shows Scenario 5 yields a further reduction of inner blocks of 14.1, and 2.3 additional blocks because an expanded physical block set is available and specifically because physical programmable blocks are available. Scenario 6 results in further decrease resulting in an inner block count of 12.9 and an additional block count of 2.9 because virtual programmable blocks are not penalized.

7.2 Minimizing Total System Cost

In this section, we consider the same scenarios previously discussed but aim to minimizing the total system cost. Because the end-user is interested in cost reduction, the tool must consider the price of blocks utilized in the final solution. In block count reduction utilizing a 2-input logic block and 3-input logic block made no difference because both had a block count of 1. However, in the system price reduction a 2-input logic block is a better choice at $7.42 than the 3-input logic block at $9.05.

Figure 10 provides a breakdown of system cost based on the block classification - physical fixed-function, physical programmable, virtual fixed-function, or virtual programmable. Figure 11 indicates the cost of blocks not currently available within the physical set that an end-user needs to purchase to implement the physical system indicated by the tool. Scenarios 7 and 8 again consider block libraries in which no physical blocks are available, thus the tool tries to find the lowest cost system implementation. Although Scenario 8 does not penalize use of programmable blocks,

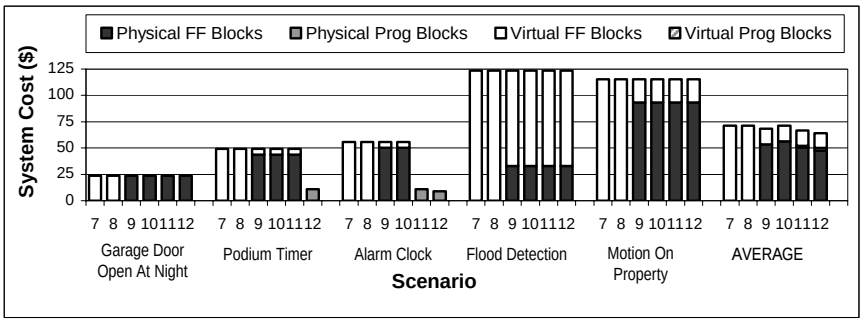

Figure 10. Cost of physical fixed function (FF) blocks, physical programmable (Prog) blocks, virtual fixed function blocks, and virtual programmable blocks.

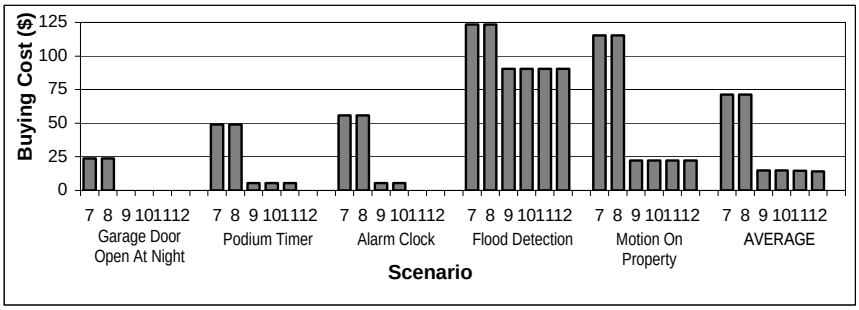

Figure 11. Additional inner block cost required to implement the corresponding system

there was no cost benefit in utilizing programmable blocks. Both Scenarios 7 and 8 resulted, on average, in an inner block cost of $71.29. Scenarios 9 and 10 include a library of fixed function blocks, where the end-user assigns larger weights to virtual blocks with virtual programmable block receiving an even higher weight. Again, there was no cost benefit in utilizing programmable blocks with both scenarios resulting in a total system cost of $71.20, and a cost of $14.80 for virtual blocks. Scenarios 11 and 12 included a library of both fixed function and programmable blocks thus resulting in inner block costs of $58.62 and $64.13 and an additional block cost of $9.31 and $14.12 respectively.

Overall, our technology mapping and optimization enables end-users to successfully design a system with existing blocks or with minimal additional blocks required. Additionally, our optimization tool is effective in reducing the size of end-user designed systems and reducing system cost. On average, our tool is extremely fast, requiring only 6 second per application, executing on a 2.8 GHz Xeon computer. When the end-user selected system cost reduction as the optimization criteria, the tool yielded a 23% reduction of system cost compared to the original implementation. When the end-user selected block count minimization as the optimization criteria, on average the tool yielded system implementations requiring six virtual blocks.

8 Conclusions and Future Work

We described a technology mapping and optimization tool to aid end-users in transforming a virtual eBlock system into an optimized physical block system. The tool requires no programming or electronics experience on the end-user's part, yet provides end-users with the ability to guide the tool in producing a system optimized for size or cost based on a constrained block library. The tool presented in paper is part of a larger framework. We plan to continue to add more blocks to the eBlock set as well as to expand the tools to support customization of the communication protocol and block parameters. The blocks, combined with the tool, help end-users setup useful basic sensor-based embedded computing systems to monitor and control the end-users' environments.

9 Acknowledgments

This work is supported in part by the National Science Foundation under grant CCR-0311026. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

10 References

1. Blackwell, A., R. Hague. AutoHAN: An Architecture for Programming the Home. IEEE Symposia on Human-Centric Computing Languages and Environments, 2001.
2. Chen, K., J. Cong, Y. Ding, A. Kahng, P. Trajmar. DAG-Map: graph-based FPGA technology mapping for delay optimization. IEEE Design & Test of Computers, Volume 9, Issue 3, Sept. 1992.
3. Cong, J., Y. Ding. On area/depth trade-off in LUT-based FPGA technology mapping. IEEE TVLSI, Volume 2, Issue 2, June 1994.
4. Cong, J., Y. Ding. FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Volume 13, Issue 1, January 1994.
5. Cotterell, S., F. Vahid. A Logic Block Enabling Logic Configuration by Non-Experts in Sensor Networks. Conference on Human Factors in Computing Systems, April 2005.
6. eBlocks: Embedded Systems Building Blocks. http://www.cs.ucr.edu/~eblock
7. Francis, R. Technology Mapping for Lookup-Table Based Field-Programmable Gate Arrays. PhD Thesis, Department of Electrical Engineering, Univ. of Toronto, 1993.
8. Francis, R., J. Rose, Z. Vranesic. Technology mapping of lookup table-based FPGAs for performance. ICCAD, 1991.
9. Francis, R., J. Rose, K. Chung. Chortle: a technology mapping program for lookup table-based field programmable gate arrays. DAC, 1990.
10. Gregory, D., K. Bartlett, A. de Geus, G. Hachtel. Socrates: A System for Automatically Synthesizing and Optimizing Combinational Logic. DAC, 1986.
11. Joyner, W. H., L.H. Trevillyan, D. Brand, T. A. Nix, S. C. Gundersen. Technology Adaptation in Logic Synthesis. DAC, 1986.
12. Kelleher, C., R. Pausch. Lowering the Barriers to Programming: A taxonomy of programming environments and languages for novice programmers. ACM Computing Surveys (CSUR), Vol. 37 Issue. 2, 2005.
13. Keutzer, K. DAGON: technology binding and local optimization by DAG matching. DAC, 1987.
14. Kirkpatrick, S., C. Gerlatt, M. Vecchi. Optimization by Simulated Annealing, Science 220, 671-680, 1983.
15. Lysecky, S., F. Vahid. Automated Application-Specific Tuning of Parameterized Sensor-Based Embedded System Building Blocks. UbiComp, 2006.
16. McNerney, T. Tangible Programming Bricks: An Approach to Making Programming Accessible to Everyone. S.M. Thesis, MIT Media Lab, 2000.
17. Morgan, R. Building an Optimizing Compiler Butterworth-Heinemann, 1998.
18. Smarthome, http://www.smarthome.com, 2006.
19. Vahid, F., S. Cotterell, S. Bakshi. eBlocks: Embedded Systems Building Blocks. Harvard Business School Business Plan Contest, 2004.
20. Wyeth, P. and H. Purchase. Tangible Programming Elements for Young Children. Extended Abstract CHI, 2002.