Addressing the Tradeoff Between Standard and Custom ICs in System Level Design

Jay K. Adams, Donald E. Thomas

EDRC 18-28-92

Addressing the Tradeoff Between Standard and Custom ICs in System Level Design *

Jay K. Adams and Donald E. Thomas

February 1992

Abstract

Digital design at the system level considers the implementation of a system with some mix of standard ICs, custom ICs, and software. In the early stages of design the system is described by a set of descriptions which may include software, hardware behavioral descriptions, and specifications for standard parts. The implementation implied by the initial set of descriptions, however, may not meet system performance goals (i.e. cost, throughput, physical size). The challenge of the early design stages, then, is to rework the set of descriptions into one whose implementation meets the performance goals. One possibility is to consider designing custom ICs for some parts of the system. This may be an attractive alternative when only a subset of the functionality of a standard IC is needed by the system, or when the standard IC implementation represents a poor use of PC board space (i.e. many SSI or MSI parts). This paper formalizes the tradeoff between using a custom IC and using standard ICs to implement part of a system. The new design tool described in this paper brings together system-level and behavioral-level synthesis paradigms and is capable of designing microprocessor-based computer systems which include an ASIC. Effective use of the ASIC's gate capacity and I/Os results in designs which require as little as 62% of the PC board area needed by designs with no ASIC.

I Introduction

Digital design at the system level considers the implementation of a system with some mix of standard ICs, custom ICs, and software. In the early stages of design the system is described by a set of behavioral descriptions which may include software (using a programming language), hardware (described behaviorally), and specifications for standard parts (e.g. memory: 32K by 8). The implementation implied by the initial set of descriptions, however, may not meet system performance goals (i.e. cost, throughput, physical size). The challenge of the early design stages, then, is to rework the set of descriptions into one whose implementation meets the performance goals. Several possibilities exist: re-forming processes, reconsidering which processes should be implemented in hardware and which should be implemented in software, repartitioning the hardware, and designing custom ICs for some parts of the system.

Even though the initial description of the system suggests the logical structure of abstract parts, the structure of the physical parts in the implementation may be quite different. Two abstract

*This work has been supported by the Engineering Design Research Center, a NSF Engineering Center

parts may be combined when a single physical part subsumes their functions; a single abstract part may be decomposed when no standard part performs its function; or the abstract part may be implemented as part of a custom IC. The last situation may arise when only a portion of a standard part's functionality is needed, thus it could be more efficiently implemented using an ASIC. It may also arise when the standard part alternative represents a poor use of PC board space; such is often the case for interface or "glue" logic.

In this paper we begin to explore the issues involved in choosing between standard and custom ICs for part of a system. By combining and extending the capabilities of two existing digital design tools, the System Architect's Workbench (SAW) [1] and the MICON System [2] [3], we have created a tool capable of addressing these issues. The new tool is able to design single board computer systems according to high level specifications which among others include: the target system cost, the available PC board area, the target power dissipation, and the sise of an ASIC (in terms of gate capacity and I/O count) that may be used in the design. The ASIC may be used to implement a number of functions that would otherwise require the use of standard parts. The result is the design of a processor board (a part list, a description of the ASIC behavior, and a net list) that meets the specifications.

The new design tool brings together two synthesis paradigms, system-level synthesis and component-level behavioral synthesis. Figure 1 shows how the system-level and component-level synthesis tools interact. At various points in its design process the system-level synthesis tool must decide between using one or more standard parts and incorporating their behavior into an ASIC. In order to make a decision the system-level tool provides specifications for a new component to the component-level synthesis tool. The component-level synthesis tool creates a component that meets the specifications and reports to the system-level synthesis tool the characteristics (e.g. circuit area) of the new component. The system-level tool then makes its decision based on the characteristics of the new-component and those of the standard parts.

The specifications for a new component consist of a behavioral description, clock frequency requirements, timing constraints, and *functional parameters*. Functional parameters describe how the component is to be used in the system and may implicitly or explicitly declare that some of the functionality included in the behavioral description is not used. The functional parameters, then, allow the component-level synthesis tool to make use of knowledge about how the component is to be used in the system.

The exchange of information between the two tools allows the component-level synthesis tool to specialize the component for its intended use while allowing the system-level synthesis tool to reason about the exact cost of implementing the new component. It is the job of the component-level tool to efficiently synthesize a component which meets the specifications; and the job of the system-level tool to determine whether or not the new component should be included in the system.

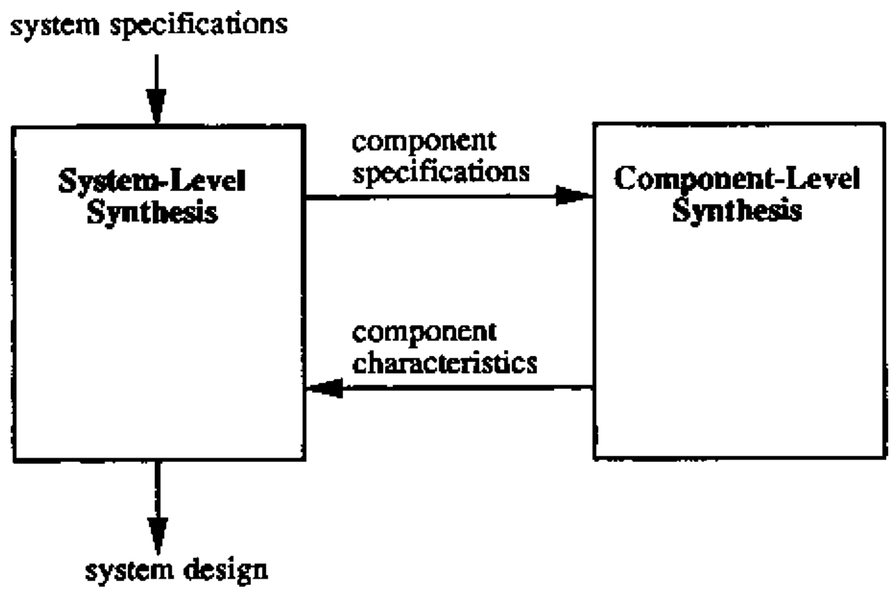

Figure 1: The interaction between system-level and component-level synthesis tools.

As an example of the interaction between the two synthesis tools, suppose that the system-level tool needs to implement a serial I/O controller. It knows that the Intel 8251 will suffice. In order to find out how much circuit area would be used if the serial I/O controller were incorporated into the ASIC, the system-level tool passes a behavioral description of the i8251 to the component-level synthesis tool along with the clock frequency requirements (e.g. maximum baud rate). Suppose further that the i8251 will always be operated in asynchronous mode. The system-level tool could pass that information along in the form of a functional parameter. The component-level tool would then synthesize an ASIC version of the i8251 which only supports asynchronous mode and report its characteristics back to the system-level tool. The system-level tool would then choose between using the i8251 and including its functionality on an ASIC.

The system-level synthesis tool may also choose to include the functionality of an SSI or MSI part on the ASIC. In that case we assume that a macro exists for the ASIC version of the SSI or MSI part and that no synthesis is necessary; the system-level tool simply makes its choice based on the characteristics of the macro and those of the SSI or MSI part.

Given that the component-level and system-level synthesis tools exist, only two issues remain: how the component-level synthesis tool can make use of functional parameters to customize a design for a given application, and how the system-level synthesis tool chooses between the new component and standard ICs.

The following section briefly discusses previous research along these lines. Section III describes how functional parameters axe handled by the component-level synthesis tool. Section IV describes how the system-level synthesis tool selects between the standard and custom IC alternatives. Section V describes the results of experiments with the new design tool. Finally, Section VI offers some conclusions as well as indications of future directions.

II Related Research

System level synthesis has been the focus of the MICON project [2] [3] for several years. MICON is a knowledge-based expert system capable of producing designs for single-board computer systems based on system-level goals such as the total amount of PC board area. system cost, and power dissipation. MICON, however, is currently only able to use standard, off-the-shelf components in the systems it designs. This paper will show how adding a high-level synthesis engine to a system-level synthesis tool such as MICON allows it to design systems which include custom ICs.

High level synthesis (register-transfer level synthesis from a behavioral description) has been an active area of research for some time. The System Architect's Workbench (SAW) [1], for instance, is able to design an ASIC according to a high-level behavioral description. High-level synthesis systems are able to take into account performance goals such as timing constraints [4]. clock speed, and circuit area. They are also able to partition functionality among several physical components [5] [6]. Current high-level synthesis techniques are inappropriate for system-level design, however, because of their inability to reason about complex physical components (e.g. memories, microprocessors). This paper builds on previous high level synthesis work by incorporating it into system-level synthesis and adding partial evaluation as a way of synthesizing a subset of a behavior.

Partial evaluation. especially as it applies to compiling and compiler generation, has also been an active topic of research [7]. Recently, Berlin [8] and Weise [9] reported the use of partial evaluation as a way of specialising scientific code for a given situation (e.g. turning a program for solving the n-body problem into a more efficient one for solving the 3-body problem). We apply partial evaluation in order to produce a specialized component from a general behavioral description. Although the goal of the Berlin and Weise use of partial evaluation is similar to ours, partial evaluation, to our knowledge, has never been used to specialize a hardware description prior to synthesis.

III Functional Parameters

The purpose of functional parameters is to allow the component-level synthesis tool to make use of knowledge about how the component is to be used in the system. Functional parameters appear in a behavioral description as constant declarations. The values that are declared to be constant may either be inputs to the component (e.g. a pin on the component that means "operate in mode A" or "operate in mode B") or "flags" in the behavioral description (i.e. a variable that means "is function X required"). For example, if the behavioral description described an up/down counter, a functional parameter might imply that the "count up" input is always false. In that case, the parts of the behavioral description that deal with counting up could be eliminated.

Our approach to making use of functional Parameters relies on partial evaluation to eliminate parts of a behavioral description which are not need. The goal of partial evaluation, as it is implemented here, is to remove paths in the control flow that cannot be executed. Specifically, we wish to determine, for every n-way branch in the behavioral description, which alternatives cannot be taken in light of functional parameters.

First, we define an n-bit *partial constant*, $\underline{P}$, to be $p_1p_2...p_n$ where $p \in \{0, 1, -\}$. We say that a constant, $\underline{A} = a_1a_2...a_n$ where $a \in \{0, 1\}$, is *consistent* with a partial constant, $\underline{P}$, if and only if for every i for which $p_i \in \{0, 1\}$, $a_i = p_i$. Intuitively, a partial constant represents a binary number in which some of the bits are constant and some are variable.

Let $C(\underline{P})$ be the set of all constants that are consistent with $\underline{P}$. If $\underline{P}$ is an m-bit partial constant and $\underline{Q}$ is an n-bit partial constant, then a partial evaluation, PE, of an n-bit function applied to $\underline{P}$ is is defined to be

$$\mathrm{PE}(f, \underline{P}) = \underline{Q} \text{ iff } \forall \underline{A} \in C(\underline{P}) : f(\underline{A}) \in C(\underline{Q})$$

This definition guarantees that a PE operator preserves the semantics of the function. It says nothing about the ability of a PE to propagate constant information.

In order to make use of the constants in $\underline{P}$ we define PE so that "1" OR'ed with "1," "0," or "-" yields a "1"; and "0" AND'ed with "1," "0," or "-" yields a "0." Using similar reasoning, we define PE to take advantage of constant bits in the inputs of any logical or arithmetic operation. Figure 2 shows examples of our PE applied to AND (&), OR (|), GREATER (>), and PLUS (+).

We assume that an n-way branch construct (analogous to a CASE or IF-THEN-ELSE statement) consists of a selector and a number of alternatives. The selector is a function and a set of inputs; and each alternative is a constant and a list of operations. The semantics of an n-way branch are such that an alternative is active when its constant equals the value of the selector function applied to the inputs. When an alternative is active, control passes from evaluating the selector function to the list of operations associated with that alternative.

Unreachable branch alternatives are removed by considering each n-way branch in the behavioral description. The selector function of the branch is evaluated with the inputs (a partial constant) to obtain the selecting partial constant. Then, any branch alternatives whose constant is not consistent with the selecting partial constant can be removed. If only one alternative is left, we remove the branch construct itself, and replace it with the operations associated with the alternative. The definition of a PE operator given above, ensures that removing branch alternatives in this way will not alter the semantics of the branch. An example is shown in Figure 3.

```
0100 | ---1   →   -1-1

010- & 0--1   →   0-0-

1-11 > 10-0   →   1 (true)

00-1 + 10-0   →   sum=1--1, carry=0
```

Figure 2: **Partial Evaluation: Even though some of the bits in the operands are unknown, some conclusions can be drawn about the result.**

```
P = 010;                                  P = 010;
Q = input1;                               Q = input1;

case ( P | Q )                            case ( P | Q )
  0:
    begin ...  end
  1:
    begin ...  end
  2:                              →         2:
    begin ...  end                            begin ...  end
  3:                                        3:
    begin ...  end                            begin ...  end
  4:
    begin ...  end
  5:
    begin ...  end
  6:                                        6:
    begin ...  end                            begin ...  end
  7:                                        7:
    begin ...  end                            begin ...  end
endcase;                                  endcase;
```

Figure 3: PE applied to "`P | Q`" results in the partial constant "`-1-`". The alternatives that are inconsistent with "`-1-`" are removed.

IV Standard versus Custom ICs

The system-level design tool must be able to choose between ASIC-based and a standard part based implementations. Our approach to this problem is similar to that used by MICON to choose among alternative standard parts[10]. The system-level synthesis tool calculates what impact each alternative will have on the physical size, dollar cost, total power dissipation, available ASIC area, and available ASIC I/Os of the evolving design. Standard ICs will contribute to the physical size, dollar cost, and power dissipation of the design but will not affect the available ASIC area or I/Os. An ASIC-based implementation, on the other hand, will affect the available ASIC area and I/Os but will not contribute to the physical size, dollar cost, or power dissipation of the design.

The actual choice is made by computing a cost for each alternative then choosing the alternative with the lowest cost. The cost is a weighted sum of the percentage of each system resource (i.e. PC board area, cost, power dissipation, ASIC area, and ASIC I/Os) consumed by the alternative. The cost is determined by

$$\text{Cost} = \sum_{\text{resources } i} W_i \frac{N_i}{T_i}$$

where N_i is the amount of resource i required by the part, T_i is the initial amount of resource i in the system, and W_i is the variable weight factor. As the design evolves, W_i changes to express how important it is to conserve resource i. W_i increases as resource i is consumed. If W_i for a particular resource becomes larger than the weight factors of the other resources, it indicates that resource i is in relatively short supply and, consequently, resource i should be conserved if possible.

The weight factor for each resource, W_i, is given by

$$W_i = \frac{T_i}{A_i + (0.01)T_i} S_i$$

where A_i is the amount of each resource that is currently available and S_i is a constant scale factor for each resource. The scale factor addresses the fact that the use of some percentage of one resource may not be comparable to the use of the same percentage of another resource. For example, it may be desirable to choose a component that requires 10% of the ASIC resources and no additional board area over one that requires 5% of the board area and no ASIC resources. While there will usually be only enough ASIC resources to implement a fraction of the entire system, there must be enough board area to implement the entire system, including parts for which no ASIC alternative exists (e.g. the CPU or the memory). Thus, we expect that the amount of ASIC resources needed to implement some function will be a larger fraction of the total than that of the board area needed to implement the same function. Experiments have shown that when the scale factors for board area, cost, and power dissipation are unity, ASIC area and ASIC I/O scale factors of 0.05-0.10 work well.

In order for ASIC I/O usage to be figured into the cost function, the system-level synthesis tool must be able to determine, at any point in the design, how many I/O's are available on the ASIC

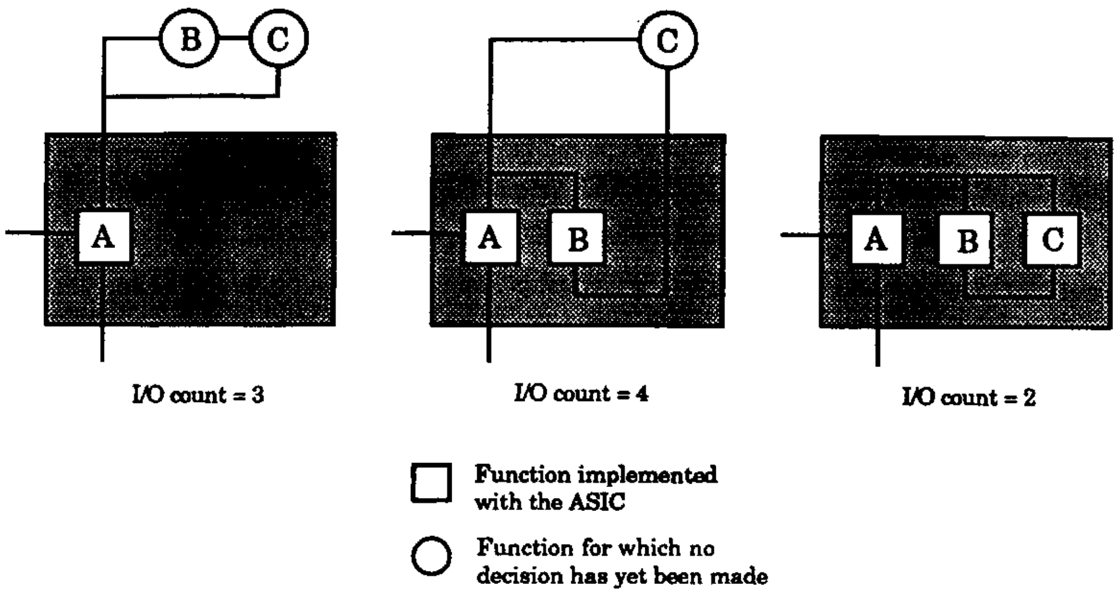
B
C
A
I/O count = 3
C
A
B
I/O count = 4
A
B
C
I/O count = 2
Function implemented with the ASIC
Function for which no decision has yet been made

Figure 4: An example of how ASIC I/Os are counted by the system-level synthesis tool.

and how many will be consumed if some function is added to the ASIC. We assume that an ASIC I/O must be used for every signal that connects an ASIC circuit to a non-ASIC circuit (i.e. I/Os are not multiplexed). At any point in the design process, the number of I/Os in use is determined by two rules:

1. Count one I/O for each net which connects to both an ASIC circuit a non-ASIC circuit.
2. If the design of any component on a net is unfinished, assume the net is connected to a non-ASIC circuit.

Rule 2 ensures that the I/O count is always pessimistic. It also ensures that, at any point in the design, the decision to use a standard IC does not impact the ASIC I/O availability. Figure 4 shows an example of how the ASIC I/O count progresses as the design evolves.

V Results

SAW and MICON enhanced and combined to implement the design tool described in the previous sections. The new tool is capable of designing single-board computer systems which may contain a single ASIC. We limit the design to a single ASIC in order to avoid the problem of partitioning functionality among ASICs. The design tool takes as input the target system cost, the available PC board area, the system I/O requirements, the target power dissipation, the gate capacity of an ASIC, and the I/O count of an ASIC. Its output is the design of a single-board computer system system, which includes a part list, a description of the ASIC, and a net list.

Partial Evaluation

Figure 5 shows the results obtained when partial evaluation was applied to several behavioral descriptions prior to synthesis. The physical size of the design is shown in terms of controller states, register bits, functional unit gates, and MUX inputs. The CDP1802 is an 8-bit microprocessor. It was synthesised for the case in which the interrupt and DMA inputs were declared to be inactive. The AM2903 is a 4-bit processor bitslice. The AM2903 was synthesised for three cases: one in which the part was to be the least significant slice, one in which the part was to be the most significant slice, one in which the part was to be an intermediate slice. The i8251 is a serial I/O interface whose behavioral description consists of three processes. The results shown are for the transmitter process only. It was synthesised for two cases: one in which the only synchronous mode was enabled and one in which only asynchronous mode was enabled. ATBI is a memory and I/O bus interface designed for an 80386-based workstation. ATBI may or may not include a write buffer. It was synthesized for the two cases in which a single write buffer entry is and is not required. All descriptions were also synthesised without using partial evaluation so that hardware would be created for all of the behavior in the original description. In all examples, partial evaluation was able to reduce the size of the resulting hardware by eliminating those parts of the behavior which are not needed.

Standard versus Custom ICs

The new tool was used to produce designs for an 80386-based processor board in a variety of scenarios. The scenarios differ only in the specifications supplied for the available PC board area, ASIC gate capacity, and ASIC I/O count. The design of the processor board is such that the keyboard controller and the serial I/O (SIO) controller may be implemented either by a standard part or as part of an ASIC. Also, the memory bus interface may be implemented either as part of the ASIC or with standard SSI and MSI TTL parts. In addition to these subsystems, some of the processor's glue logic may also be implemented with the ASIC, eliminating the need for it to be realised with discrete TTL parts.

Early experiments revealed that better results are obtained when the decisions were made in order of their impact on the final board area. Since implementing the bus interface using the ASIC rather than discrete TTL components results in the greatest reduction in board area, the system-level tool is programmed to decide how the bus interface is implemented before considering the other subsystems. It then makes decisions about how the keyboard controller and SIO controller (in that order) are implemented.

Over the many scenarios that were attempted several tendencies, with respect to selecting a custom rather than a standard IC, were observed. The tool chose to implement the bus interface with the ASIC in all cases in which the ASIC had sufficient area and I/Os. This is clearly due to

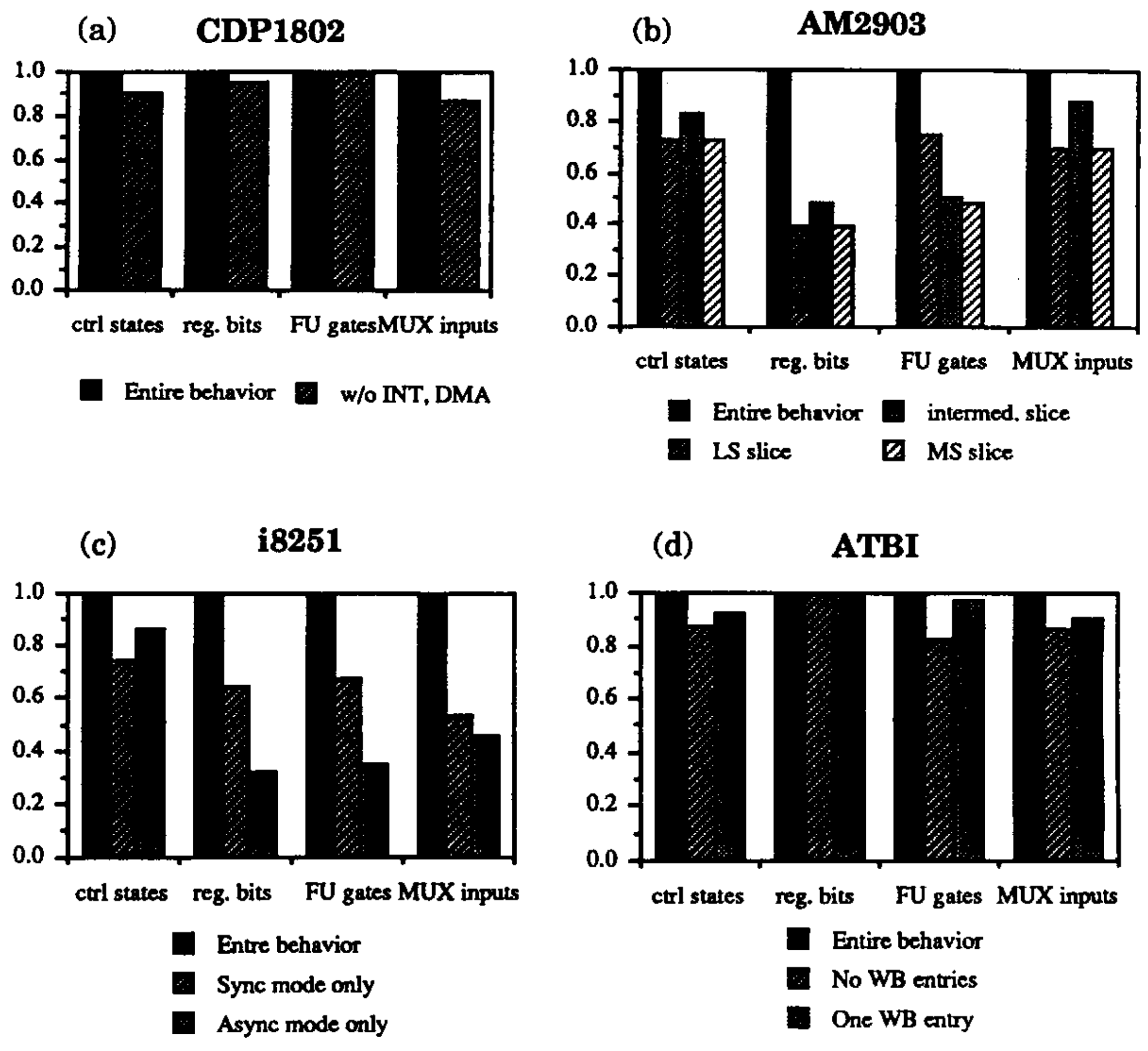

Figure 5: Partial evaluation results. The amount of hardware produced is normalized to the sise of the "entire behavior" case.

the fact that implementing the memory bus interface with the ASIC saves a considerable amount of board area. The keyboard controller was usually realized with the ASIC if enough area and I/Os were available. Since the keyboard controller shares many I/O's with the bus interface, adding it to the ASIC requires using only a few additional ASIC I/Os (provided that the bus interface is also being implemented by the ASIC). The tool chose to use the ASIC for the SIO controller only when ASIC area was in ample supply, either because a large ASIC was specified or because the ASIC was not used to implement the bus interface or keyboard controller. This too was expected since, due to the small physical size of the standard component that may be used to implement the SIO controller, it does not provide as much opportunity for saving board area.

We believe that in general our tool will always choose an ASIC implementation over one composed of many discrete logic parts (provided that the ASIC has sufficient gate capacity and number of I/Os) because of the amount of board area that is saved. When the choice is between using a single standard part and adding to the ASIC, the use of the ASIC does not offer as much reduction in board area. We believe that in these cases, our tool will choose to use the ASIC only if doing so does not consume a great deal of the ASIC I/Os and gates. This may be the case when, as in the case of the keyboard controller, relatively few additional ASIC I/Os are needed due to sharing I/Os with other functions implemented by the ASIC.

ASIC versus Discrete Logic

In order to observe how efficiently the new tool uses the ASIC to implement glue logic we observe how the physical size of the resulting design varies with the number of ASIC I/Os used. Early experiments showed that the amount of glue logic that could be implemented with the ASIC was primarily a function of the number of I/Os on the ASIC.

Figures 6 and 7 show how the final PC board area of the system varies with the number of ASIC pins used when the ASIC gate capacity is fixed at 8000 and 12000 gates, respectively. Without using an ASIC, the system would require $105\,\text{in}^2$ of board area. In some cases the board area does not change even though more ASIC pins are used. Such is the case in Figure 6 when the number of ASIC pins goes from 89 to 97 and in Figure 7 when the number of pins goes from 97 to 108. This happens when the ASIC is used to implement one or more logic gates which would otherwise be part of some other multiple-gate package. For instance, suppose a design contains four NAND gates. They may all be implemented either with a single 7400 or an equivalent part. If, however, the ASIC were used to implement three of the NAND gates, a 7400 would still be needed for the remaining NAND gate and no improvement in board area would be seen.

Closer examination of the results reveals that parts which represent a high board area to I/O count ratio (the amount of board area needed to implement the function with discrete parts versus the number of additional ASIC I/Os needed to implement the function with the ASIC) are likely

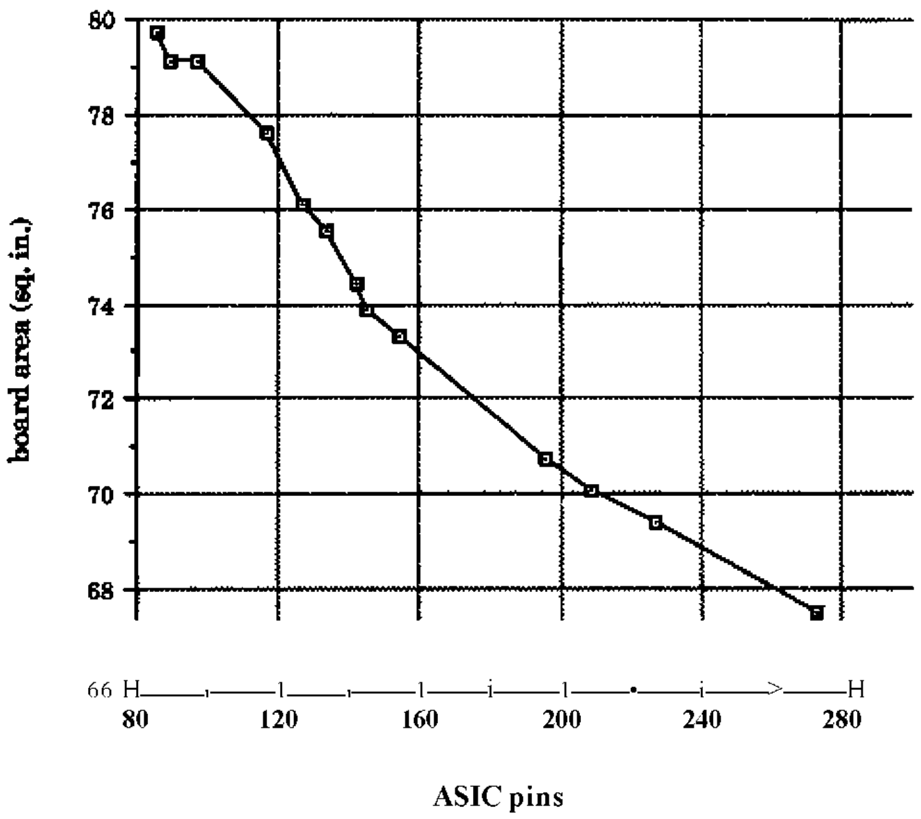

Figure 6: Board area obtained versus ASIC pins used with ASIC area fixed at 8000 gates.

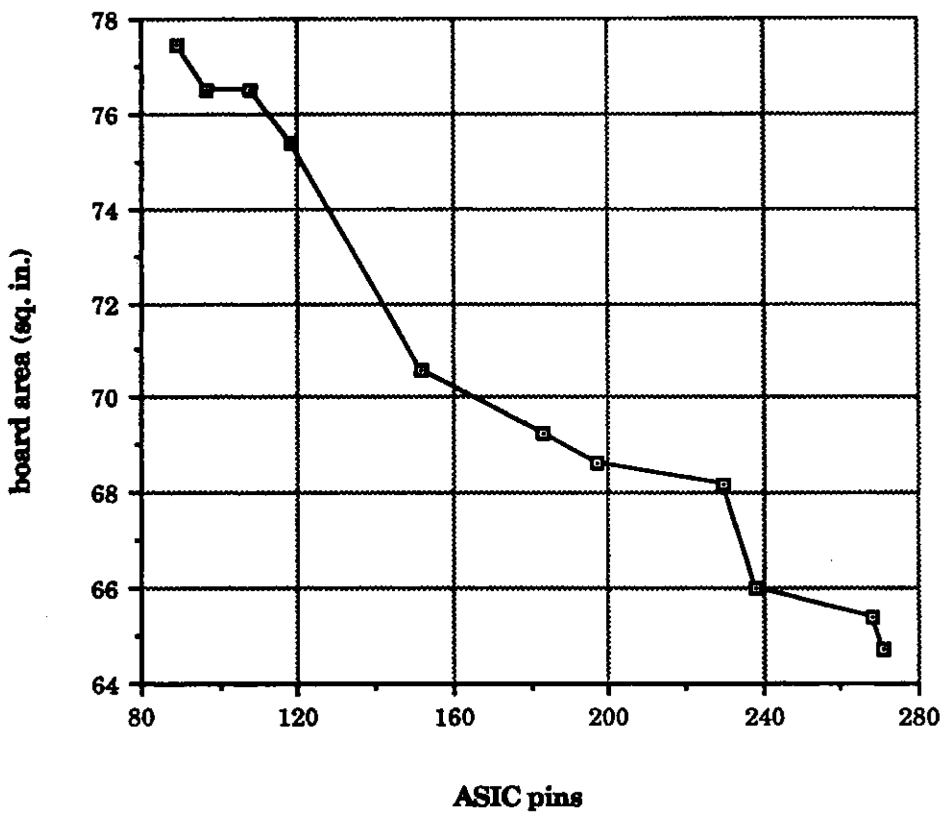

Figure 7: Board area obtained versus ASIC pins used with ASIC area fixed at 12000 gates.

to be implemented with the ASIC even when the availability of ASIC I/Os is limited. Individual logic gates are examples of functions which exhibit a high board area to I/O count ratio especially when they share I/Os with other functions implemented by the ASIC. Only when ASIC I/Os are abundant are parts which represent a relatively low board area to I/O count ratio implemented with the ASIC. This phenomenon is apparent in the graph shown in Figure 6. The difference in board area between using 100 and 150 ASIC I/Os is approximately 5.5in^2 while the difference between using 200 and 250 ASIC I/Os is only about 2.0in^2. A similar observation could be made about the graph shown in Figure 7.

VI Conclusions

The design tool described in this paper a formalizes the decision between standard and custom ICs. Functional parameters allow a system-level synthesis tool to communicate knowledge about a component's use to a component-level synthesis tool. Furthermore, partial evaluation allows a component-level behavioral synthesis tool to use that knowledge to produce more efficient designs. By combining the functionality of complex standard ICs and with that of low-level SSI and MSI ICs, the new design tool is able to specify an ASIC which subsumes the function of many standard ICs.

This work offers some opportunity for improvement and extension. Considering the use of multiple ASIC's, for instance, would allow more of the system's functionality to be integrated into custom ICs. However, it would require the ability to effectively partition the functionality.

Partial evaluation in high-level synthesis provides a way to specialise hardware in much the same way it does certain types of computer programs. This capability could allow digital designers (be they human or automatic) to tailor a fairly general hardware description to many situations. The result could be less time spent developing hardware descriptions and the ability to use high-level abstractions in hardware descriptions without compromising efficiency.

The ideas presented in this paper might also be extended to address the issue of hardware versus software implementations. If the specifications for a system include software, we could consider designing special purpose hardware to replace the software and the processor on which it runs. The choice between the software implementation and the special purpose hardware could then be made in a manner similar to that used for standard versus custom ICs.

References

[1] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, and R. L. Blackburn, *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench.* The Kluwer International Series In Engineering and Computer Science. Kluwer Academic Publishers, 1990.

[2] W. P. Birmingham, A. P. Gupta, and D. P. Siewiorek, "The MICON System for Computer Design," in *Proceedings of the 26th Design Automation Conference*, IEEE Computer Society and ACM-SIGA, 1989.

[3] A. P. Gupta and D. P. Siewiorek, "M1: A Small Computer System Synthesis Tool," in *6th IEEE Conference on AI Applications Proceedings*, 1990.

[4] J. A. Nestor, *Specification and Synthesis of Digital Systems with Interfaces.* PhD thesis, Carnegie-Mellon University, April 1987.

[5] E. D. Lagnese, "Architectural Partitioning for System Level Synthesis of Integrated Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, pp. 847–860, July 1991.

[6] R. Gupta and G. De Micheli, "Partitioning of Functional Models of Synchronous Digital Systems," in *Digest of Technical Papers: IEEE International Conference on Computer-Aided Design*, (Santa Clara), November 1990, pp. 216–9.

[7] D. Bjorner, A. P. Ershov, and N. D. Jones, eds., *Partial Evaluation and Mixed Computation.* North-Holland, 1988.

[8] A. A. Berlin, "A Compilation Strategy for Numerical Programs Based on Partial Evaluation," Master's thesis, M.I.T., 1989.

[9] A. A. Berlin and D. Weiss, "Compiling Scientific Code Using Partial Evaluation," *Computer*, pp. 25–37, December 1990.

[10] A. P. Gupta, "Private Communication," 1991.