Using Estimates from Behavioral Synthesis Tools in Compiler-Directed Design Space Exploration *

Byoungro So, Pedro C. Diniz, and Mary W. Hall
University of Southern California / Information Sciences Institute
4676 Admiralty Way Suite 1001
Marina del Rey, California 90292
{bso,pedro,mhall}@isi.edu

ABSTRACT

This paper considers the role of performance and area estimates from behavioral synthesis in design space exploration. We have developed a compilation system that automatically maps high-level algorithms written in C to application-specific designs for Field Programmable Gate Arrays (FPGAs), through a collaboration between parallelizing compiler technology and high-level synthesis tools. Using several code transformations, the compiler optimizes a design to increase parallelism and utilization of external memory bandwidth, and selects the best design among a set of candidates. Performance and area estimates from behavioral synthesis provide feedback to the compiler to guide this selection. Estimates can be derived far more quickly (up to several orders of magnitude faster) than full synthesis and place-and-route, thus allowing the compiler to consider many more designs than would otherwise be practical. In this paper, we examine the accuracy of the estimates from behavioral synthesis as compared to the fully synthesized designs for a collection of 209 designs for five multimedia kernels. Though the estimates are not completely accurate, our results show that the same design would be selected by the design space exploration algorithm, whether we use estimates or actual results from place-and-route, because it favors smaller designs and only increases complexity when the benefit is significant.

Categories and Subject Descriptors

B.6.3 [**Design Aids**]: Automatic Synthesis, Optimization; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Design, Experimentation

*Funded by the National Science Foundation (NSF) under grant CCR-0209228 and the Defense Advanced Research Project Agency (DARPA) under contract #F30602-98-2-0113

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2003, June 2–6, 2003, Anaheim, California, USA.
Copyright 2003 ACM 1-58113-688-9/03/0006 ...$5.00.

Keywords

Synthesis Techniques for Reconfigurable Computing, Design Space Exploration, Rapid Prototyping, Field-Programmable-Gate-Array, High-level Synthesis

1. INTRODUCTION

The extreme flexibility and the growing capacity of Field Programmable Gate Arrays (FPGAs) has made them the medium of choice for fast hardware prototyping and a popular vehicle for the realization of custom computing machines. These custom computing machines can achieve substantial performance gains over traditional computing architectures by exploiting vast amounts of instruction-level parallelism, customizable and reconfigurable I/O bandwidth or simply by implementing directly in hardware specialized and complex functions that would otherwise takes thousands of processor instructions.

Mapping computations to FPGA-based architectures, however, is a lengthy and error prone process. Programmers must assume the role of hardware designers in bridging the semantic gap between high-level programming languages such as C and hardware-oriented programming languages such as VHDL. For this purpose, we have developed DEFACTO, an automated system for mapping high-level algorithm descriptions written in C to FPGAs, based on a collaboration between a parallelizing compiler and high-level synthesis tools [4]. Using several code transformations, the system optimizes a design to increase parallelism and utilization of external memory bandwidth. Because of the complexity of synthesis, it is difficult for compiler analysis to predict *a priori* the performance and space characteristics of the resulting design. Thus, like a human designer, the compiler must engage in an iterative process called *design space exploration*, of synthesizing a design, examining the results, and modifying the design to trade off between performance and area. Completely synthesizing a design is prohibitively slow (hours to days). Instead, our system exploits estimation from behavioral synthesis to determine specific hardware parameters with which it can quantitatively evaluate the application of a transformation to derive an optimized and feasible implementation of the computation. Behavioral synthesis estimates can be derived far more quickly (up to several orders of magnitude faster) than full synthesis and place-and-route, thus allowing the compiler to consider many more designs than would otherwise be practical.

In previous work, we described the automated design space

exploration algorithm implemented in DEFACTO, and presented estimates from behavioral synthesis, demonstrating that for five multimedia kernels, automated design space exploration selected a near-optimal design among those considered, while only searching on average 0.3% of the search space [8]. This prior work relied strictly on behavioral synthesis estimates for guiding design selection, under the assumption that these estimates were accurate performance and area predictors of FPGA designs. In this paper, we evaluate the accuracy of behavioral synthesis estimation, when used in conjunction with our design space exploration algorithm. By examining a total of 209 automatically-generated design alternatives for five multimedia kernels, we conclude that, while not fully accurate, behavioral synthesis estimates nevertheless lead the compiler algorithm to the same design as if the compiler had access to the actual implementation data from fully synthesizing all of the alternative implementations. While there are a few systems that automatically synthesize hardware designs from C specifications (*e.g.,* [12]), to our knowledge this is the first implemented system that combines parallelizing compiler technology and behavioral synthesis estimation, in an automated design space exploration algorithm.

The remainder of the paper is organized as follows. In the next section we discuss the analyses and transformations our algorithm uses. In section 3, we describe the optimization goal of our design space exploration algorithm in mapping loop nest computations to hardware. In section 4, we present the interface between our compiler and common synthesis tools. In section 5 , we present experimental results for the application of this algorithm to five multimedia kernels. We survey related work in section 6 and conclude in section 7.

2. SYSTEM OVERVIEW

To automate design space exploration, we must define a set of transformations to be applied and metrics to evaluate specific candidate designs. We now describe the set of transformations, and we discuss the metrics in the next section. Fig. 1 shows the steps of automatic application mapping in the DEFACTO compiler. We leverage the Stanford University Intermediate Format (SUIF) system, and augment its analyses with transformations for FPGA-based systems.

The design of the DEFACTO system is largely motivated by the observation that the potential of synthesis tools can be greatly improved by employing state-of-the-art parallelizing compiler analyses. In particular, data dependence analyses allow compilers to understand data access patterns in multi-dimensional arrays, absent from current synthesis tools. Using data dependence information, the DEFACTO compiler, can successfully identify multiple accesses to the same array locations across iterations of multi-dimensional loop nests. This analysis can be used to identify opportunities for exploiting parallelism, eliminate unnecessary memory accesses, guide iteration reordering transformations, and optimize mapping of data to external memories. In contrast, synthesis tools for the most part only perform optimizations on scalar variables, and within the body of a loop, and not across loop iterations. On the other hand, synthesis tools offer capabilities not present in conventional technology, including scheduling, allocation, and binding of hardware resources. To this end, the DEFACTO system combines these two complementary technologies in an integrated system.

In this paper, we focus on mapping a loop nest computation to a single FPGA with multiple external memories. The compiler techniques employed in our system target the enormous potential for parallelism in FPGA-based systems, since the number of specific resources is not fixed, and we customize the implementation to increase utilization of available memory bandwidth. We exploit instruction-level and memory parallelism using *loop unrolling* (for innermost loops) and *unroll-and-jam* for outer loops. *Unroll-and-jam* involves unrolling one or more loops in a nest and fusing copies of the inner loop together [2]. As a result, the logical operations and their corresponding operands in the loop body are replicated and exposed to behavioral synthesis optimizations. *Scalar replacement* is used to reduce the number of memory accesses by replacing certain array references with temporary scalar variables that will be mapped to on-chip registers by behavioral synthesis. As such, the subsequent references to the same array element need not access memory. For the remaining memory accesses after scalar replacement, *custom data layout* [9] distributes the accesses across multiple memories, according to their access pattern, such that the accesses can be performed in parallel across multiple memories. Thus, it effectively increases memory bandwidth utilization. Next the compiler translates the SUIF intermediate code into behavioral VHDL.

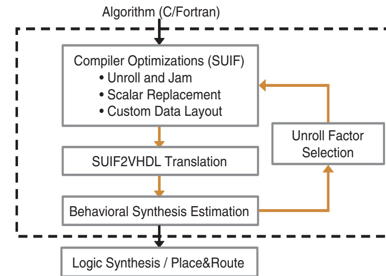

Figure 1: Design Flow

The design space exploration algorithm involves an iterative evaluation to find the best unroll factors for loops in a loop nest computation. For each fixed set of unroll factors, unroll-and-jam, scalar replacement and custom data layout are performed. This design space exploration strategy is fast, since it bypasses logic synthesis and place-and-route as much as possible. In addition, it does not explore all possible designs in a brute-force way. The optimization search considers only a very small portion of the possible unroll factors, because it is guided by the set of metrics described in the following section.

3. OPTIMIZATION STRATEGY

In this section, we summarize the design space exploration algorithm, most of which was presented in more depth in [8]. Using the previously described code transformations, the optimization criteria for mapping a loop nest computation to an FPGA with multiple external memories are as follows:

1. The design must not exceed the capacity constraints of the system.
2. Minimize the execution time.
3. For a given performance, minimize space usage.

The motivation for the first two criteria is obvious, but the third criterion is also needed for several reasons. First, if two

designs have equivalent performance, the smaller design is more desirable, in that it frees up space for other uses of the FPGA logic, such as to map other loop nests. In addition, a smaller design usually has less routing complexity, and may achieve a faster clock rate, which typically translates into faster overall execution time and consequently less consumed energy. Moreover, the third criterion suggests a strategy for selecting among a set of candidate designs that meet the first two criteria.

Our compiler uses several metrics, specific to a particular design implementation, to guide the selection of a design. As input to behavioral synthesis, the compiler specifies a target clock rate $Clock_{targ}$, and derives from synthesis estimates, as discussed in Section 4, the following metrics. *Area*(d) of design d, related to criterion 1 above, estimates overall design area and *Cycles*(d) estimates overall number of cycles, related to criterion 1. Another metric, related to criteria 2 and 3, is *Balance*(d), defined by $F(d)/C(d)$, and is the result of combining *Cycles*(d) with compiler analysis of the data accessed by each inner loop iteration of the design. Here, $F(d)$ refers to the data fetch rate, the data bits that external memory can provide per cycle, thus capturing the system's external memory bandwidth assuming the number of memories used by the design. $C(d)$ refers to the data consumption rate, the total data bits that the computation can consume per cycle. If *Balance*(d) is close to 1, both memories and FPGAs are busy. If *Balance*(d) is less than 1, the design is memory bound; if greater than one, it is compute bound.

In this paper, we introduce an additional metric *Efficiency*($d1, d2$), related to criterion 3, to compare two nearby designs $d1$ and $d2$ with respect to their relative space utilization. *Efficiency*($d1, d2$) is defined by (*Cycles*($d1$) - *Cycles*($d2$))/(*Area*($d2$) - *Area*($d1$)).

Our system applies these metrics in the design space exploration algorithm to determine the appropriate unroll factors for the loops in a loop nest, eliminating the need to consider all possible unroll factors. An additional concept, a memory bandwidth *saturation point*, refers to a certain unroll factor where the data is being fetched at a rate corresponding to the maximum bandwidth of the target architecture. The data fetch rate monotonically increases as the unroll factor increases until it reaches a saturation point, beyond which, the data fetch rate does not improve. Similarly, the data consumption rate also monotonically increases as the unroll factor increases, but less than linearly due to data dependences and idle time waiting on memory accesses. From these monotonicity properties of the fetch rate and consumption rate, *Balance* also increases monotonically as the unroll factor increases until it reaches the saturation point as the data fetch rate increases faster than the data consumption rate. *Balance* decreases monotonically beyond the saturation point because the data fetch rate does not increase, but the data consumption rate is still increasing.

For an inherently compute-bound design, performance may continue to improve as unroll factors increase, but perhaps not enough to justify the increased space usage. To capture this notion, we compute *Efficiency*($d1$,$d2$) to compare a design $d1$ that uses a particular unroll factor with a design $d2$ that uses a larger unroll factor. If *Efficiency* is below a predefined threshold, our algorithm selects $d1$ over $d2$.

We rely on these metrics and monotonicity properties to limit the number of designs that must be searched by the compiler, while still meeting our optimization criteria. The overall approach increases unroll factors only when doing so will have large performance gains, and we will see in Section 5 that such a strategy is very compatible with using behavioral synthesis estimates. Even though there is some loss of accuracy due to increased cycle time as design complexity grows, the improvement in performance due to unrolling and unroll-and-jam transformations outweighs the reductions in achieved clock rate.

4. BEHAVIORAL ESTIMATION

The design space exploration approach described in this paper relies on behavioral synthesis estimates to assess the impact on the hardware designs of each of the program transformations the compiler algorithm performs. We now describe how the compiler interfaces with behavioral synthesis and uses it to guide optimizations.

4.1 Functional Interface

This interface allows the compiler to request estimates from behavioral synthesis tools for a given design and with specific resource and/or timing constraints. The compiler uses a set of internal functions to format the request using the syntax specific to the synthesis tool. The implementation then invokes the synthesis tool in batch mode and parses the results the tool generates in the form of report files into data structures. The interface then allows the compiler to inspect the data structures to extract data about the estimates via a set of functions. This interface is currently operational for the Mentor Graphics' Monet™ tool (MT R44) and for the Synopsys Behavioral Compiler™ (BC v2000.05).

Although the interface supports a much richer set of resource constraints, in the current implementation of the design space exploration algorithm the compiler only sets the target clock rate, $Clock_{targ}$, for the various hardware processes along with the maximum allowed clock overhead. Behavioral synthesis returns *C-steps*(d), the number of control cycles per iteration each loop, and $Area_{est}(d)$, the estimated area of the design, in terms of number of slices.

The interface also exports a wealth of information about the synthesized design not exploited by the the current design space exploration algorithm. For example, the estimated clock rate $Clock_{est}$ reported by behavioral synthesis can differ substantially from $Clock_{targ}$ requested by the compiler. An excessively large discrepancy between these two metrics might suggest a design that is stressing the capacity of the target FPGA or simply exercising a poor algorithmic implementation of either the design, the tool or both. The design exploration algorithm could potentially observe this gap and react accordingly by adjusting the aggressiveness of its internal search algorithm, as for example, reducing the maximum unrolling amount explored.

4.2 Deriving Area and Speed Metrics

Using the interface outlined above, our compiler derives quantitative metrics for the performance and FPGA area for each design. To derive *Cycles*(d), the compiler multiplies *C-steps*(d), by the number of iterations of the loops in the nest.

The derivation of *Area*(d) is complicated by two main factors. First, the estimates do not include space required by logic elements inserted due to place-and-route. Second, the space metric provided for a given design is dependent on the

target FPGA devices and on the specific component library used by the synthesis tool. In the case of our experimental set up, Monet™ does not provide Xilinx specific space metrics, but rather abstract space metrics.

To address these two shortcomings, we apply a statistical approach to estimate $Area(d)$ from $Area_{est}(d)$. Using the real space capacity of a Xilinx Virtex™ XCV1000 FPGA as $C = 12,288$ slices, we have measured the exact P&R space for 209 designs for all our experiments (described in section 5). We then derive the ratio of the the space Monet™ reports in its abstract space units to the measured space usage from place-and-route. Assuming a random *normal* distribution for the space-ratio (not the real area itself), we can then compute the mean ratio range for a given confidence interval, say $100(1-\alpha)\%$. Using the bounds of the confidence interval, the compiler can then compute what the real area bounds should be based on the estimated space value (in the abstract measure). In reality, we can take advantage of the fact that the statistical distribution is symmetric whereas the compiler is only interested in the lower bound for the space-ratio range. Effectively, this allows the compiler to use a value of α that yields a tighter, hence better bound, than the value used for α would suggest. A value of $\alpha = 0.2$ for a symmetric confidence interval of 80% effectively yields the asymmetric confidence interval for a value of $\alpha = 0.1$ that is a confidence interval of 90%.

For the 209 sample designs used in our experiments the sample mean of the space-ratio of Monet™ estimate to the space usage measured by place-and-route is 2.715 and the sample variance is 0.232. For a 90% confidence interval this ratio's lower bound is 2.719 and thus the 12, 288 Virtex™ slices translate into 33, 411 as estimated by Monet™.

5. EXPERIMENTAL RESULTS

We now present experimental results for the application of our design space exploration algorithm to a set of five multimedia kernels. The goal of these experiments is twofold. First, we compare for each design the performance estimates from behavioral synthesis with actual performance obtained from logic synthesis and place-and-route. Second, we determine if our design space exploration algorithm would have selected the same design if it had accurate synthesis data rather than estimates.

5.1 Target Applications

In this experiments, we use five multimedia kernels; namely,

- Finite Impulse Response (FIR) filter: integer multiply-accumulate over 32 consecutive elements of a 64 element array.
- Matrix multiply (MATMUL): integer dense matrix multiplication of a 32-by-16 matrix by a 16-by-4 matrix.
- String pattern matching (PATTERN): character matching operator of a string of length 16 over an input string of length 64.
- Jacobi iteration (JACOBI): 4-point stencil averaging computation over the elements of an array.
- Sobel (SOBEL) edge detection: 3-by-3 window Laplacian operator over an integer image.

Each application is written as a standard C program where the computation is a single loop nest. There are no pragmas, annotations or language extensions describing the hardware implementation.

5.2 Methodology

We applied the design space exploration algorithm described in section 3 to select a balanced and efficient design for each of the application kernels outlined above. In these experiments we target a Xilinx Virtex™ XCV1000 with 4 external memories, as in the Annapolis WildStar™[1] board. In this platform, read and write operations to external memory require 7 and 3 clock cycles respectively. As part of the design space exploration the compiler derives a set of designs in behavioral VHDL, which is then synthesized by Monet™.

To compare the Monet™ estimates for each design with the real implementation metrics we performed logic synthesis with Mentor's LeonardoSpectrum™ and place-and-route with Xilinx Foundations tool set. In these experiments we also observe the impact of choosing different target clock rates in the quality of the designs. This variation in target clock rates will allow us to verify the sensitivity of our design exploration algorithm to the aggressiveness of the estimation for different clock rates and its impact on the ability of the algorithm in choosing the correct design. For these experiments target clock rates 25MHz and 40MHz, which are a reasonable targets for the Virtex™ parts in our target platform. The target clock was provided to behavioral synthesis and logic synthesis as a parameter, and both tools were directed to optimize for both performance and area.

In all, we produced behavioral synthesis results for a total of 364 points in the design space. The full place-and-route results were obtained for 209 of these points. Most of the remaining points are too large to fit within the capacity of the Virtex™ parts, and so logic synthesis does not attempt to produce a design. For seven of the points associated with PATTERN, there is an incompatibility between the output of behavioral synthesis and the expected input for logic synthesis, which prevents logic synthesis from producing a correct design. Thirteen points that are on the boundary of the Virtex™ XCV1000 capacity require more memory to synthesize than the memory available in the machine that produced our synthesis results.

5.3 Results

The first set of results, depicted in Fig. 2 and Fig. 3, compare the estimates produced by behavioral synthesis and place-and-route, respectively, for FIR, which is a 2-deep loop nest. Since the design space exploration algorithm evaluates designs resulting from different unroll factors, these results are presented as a function of unroll factors for FIR's inner and outer loops. Each curve in Fig. 2 shows results for a fixed unroll factor for the outer loop, and each point on a curve represents a different unroll factor for the inner loop. As we increase the unroll factors, the amount of available parallelism increases dramatically, up to the *saturation point*. For larger unroll factors, instruction-level parallelism may improve but memory parallelism will not.

To minimize the number of points in the search space that must be considered by the design space exploration algorithm, our compiler relies on the observation that increased parallelism, and consequently performance improvement, is monotonic as the unroll factor for a loop is increased [8]. Monotonicity of performance improvement as a function of unroll factors is clearly demonstrated in both the estimated results in Fig. 2 and the actual performance in Fig. 3, except the outer unroll factor one where there is so little re-

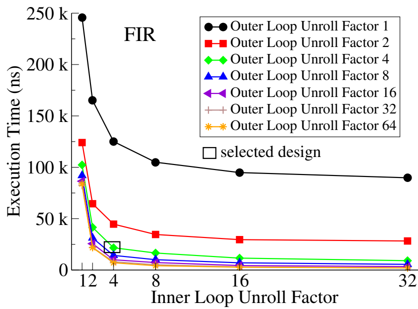

Figure 2: Estimated performance.

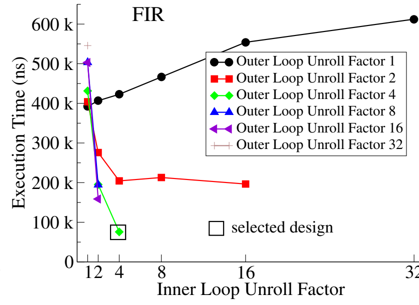

Figure 3: Achieved performance.

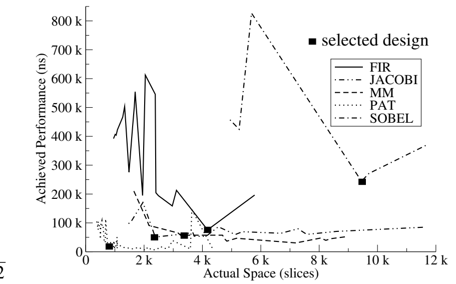

Figure 4: 25MHz Time vs. Space.

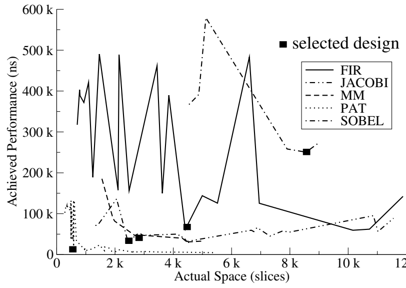

Figure 5: 40MHz Time vs. Space.

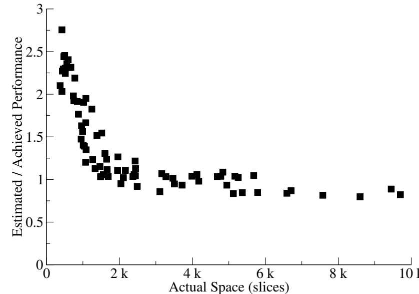

Figure 6: 25MHz Ratio vs. Space.

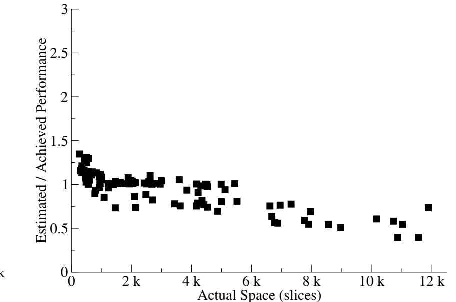

Figure 7: 40MHz Ratio vs. Space.

duction in cycles due to unrolling that the clock degradation outweighs the benefit. The *Efficiency* metric captures this behavior and prevents selection of such designs.

We compare actual performance of all five programs as a function of measured space in terms of slices in Fig. 4 and Fig. 5, for a 25MHz and 40MHz target clock rate, respectively. We see a trend that performance varies somewhat for the smaller designs, but eventually we reach a point (related to the saturation point) where performance improves at most modestly but space continues to grow. We see from these figures that, for all programs, our algorithm selects one of the best-performing designs, and the smallest design among those of comparable performance. In all programs, we can acquire better performance with the 40MHz target clock rate.

To examine the accuracy of the estimates, we plot the ratio of estimated to actual performance across the five programs in Fig. 6 and Fig. 7 for a 25MHz and 40MHz target clock rate, respectively. The Y-axis is the ratio of estimated to actual performance, so that values above 1 obtained better than expected performance, and below 1 worse than expected performance. The X-axis is measured space, so that we can see how estimation accuracy varies as the design grows more complex. The number of *c-steps* remains the same from behavioral synthesis through the final design, but there is some variation in clock rate achieved by place-and-route as the design grows due to increased routing complexity. For most of the small designs (corresponding to low values of the unroll factor), the target 25MHz was overly conservative and place-and-route was able to achieve a faster clock rate. As a result, for these designs, the estimated performance was also very conservative, yielding a ratio well above 1. These very small designs tended not to be selected by our algorithm because the number of cycles was significantly higher than those for slightly larger unroll factors, but they show that, when the clock rate is overly pessimistic, the algorithm would benefit from examining $Clock_{est}$, as discussed in Section 4. For the large designs, the target clock rate was too optimistic, but the degradation of the clock rate was at most 20% below the target 25MHz. These results reveal a discrepancy between the estimated and the actual performance most noticeable for the designs too small than for the larger designs. In Fig. 7, the accuracy of estimates improves over the accuracy for 25MHz for the smaller designs, but can be much worse for the larger designs (beyond about 50% utilization of the FPGA at 6k slices).

Overall, despite some fluctuations in the accuracy of the estimates, for either target clock rates, our algorithm selects the appropriate design because it favors smaller designs and only increases complexity if a significant benefit will be obtained.

6. RELATED WORK

We now describe current behavioral synthesis capabilities and existing approaches for design space exploration.

6.1 Behavioral Synthesis Capabilities

Current behavioral synthesis tools such as Monet^{TM} [7] or the Synopsys Behavioral Compiler^{TM} [10] allow the programmer to control the application of loop unrolling for loops with known bounds. The programmer must first convert an application to behavioral VHDL, explicitly map array variables to memories and registers, and then select the order in which the loops must execute. Next the programmer must manually determine the exact unrolling factor for each of the loops. Given the effort and interaction between the transformations, this approach to design space exploration is extremely awkward and error-prone. To the best

of our knowledge no commercial tool exploits sophisticated metrics such as *balance* to guide the application of loop transformations automatically.

6.2 Loop Transformations

Other researchers have also recognized the value of exploiting loop transformations in the mapping of regular loop computations to FPGA-based architectures. Derrien and Rajopadhye [3] describe a tiling strategy for doubly nested loops. They model performance analytically and select a tiling factor that minimizes the iteration's execution time. In their work, there is no notion of a search algorithm and no interaction with high-level synthesis.

6.3 Design Space Exploration

The PICO project at Hewlett-Packard [6] addresses the exploration of embedded systems for a target architecture that consists of an EPIC processor, a programmable hardware accelerator and a memory hierarchy. PICO decomposes the system design space into smaller design spaces for each of the major components. To address the interactions between the components, PICO uses dilation parameters thereby obtained a parameterized set of Paretto designs for each component. It then composes together, in a hierarchical fashion, enforcing that only components with the compatible dilation parameters can be merged. Rather than a single design, PICO offers for each design a selected component a set of parameterized componments.

Halambi *et al.* [5] developed a tool-kit generator system to allow designers to evaluate the impact of architectural characteristics in the mapping of an application to a SoC solution. At the core of this approach is an on-line profiling data to evaluate a given partitioning strategy. The system generates a compiler and simulator for each architecture and then uses the profiling with the application's input to understand what the performance is, and why, thereby providing value feedback to designers on how to modify the characteristics of the system.

6.4 Discussion

The work presented in this paper differs from these efforts in several respects. First, our approach takes as input a sequential application description (using a familiar programming language and not a graphical data-flow oriented specification as Cocentric[11]) and does not rely on annotations to directly control the compiler to apply specific transformations. Second, we use high-level data dependence analysis techniques and estimation to guide the application of the transformations as well as evaluate the various design points. We are able to handle multi-dimensional array variables absent in existing tools. Finally, we use a commercially available behavioral synthesis tool to complement the parallelizing compiler techniques rather than creating an architecture specific synthesis flow that partially replicates the functionality of existing tools.

7. CONCLUSION

In this paper, we have demonstrated that behavioral synthesis estimation can be effectively employed in a system that automatically explores the hardware design space of alternative compiler-generated FPGA designs. Although performance and space estimates from behavioral synthesis are not as accurate as fully synthesizing the design, we find that, when combined with our compiler algorithm for guiding design space exploration, the estimates nevertheless lead us to the best design, according to our selection criteria. The combined system quickly explores a broad range of designs, yielding solutions in a few hours that would take weeks to derive otherwise.

As technology advances increase the density of FPGA devices, tracking Moore's law for conventional logic, devices will be able to support more sophisticated functions. With the trend towards on-chip integration of internal memories, FPGAs with special-purpose functional units are becoming attractive as a replacement for ASICs and for custom embedded computing architectures. We foresee a growing need to combine the strengths of high-level program analysis techniques, to complement the capabilities of current and future synthesis tools.

8. REFERENCES

[1] Annapolis Microsystems Inc. *WildStar^TM Reconfigurable Computing Engines. User's Manual*, r3.3 edition, 1999.

[2] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in a loop. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 16(6):1768–1810, Nov. 1994.

[3] S. Derrien and S. Rajopadhye. Loop tiling for reconfigurable accelerators. In *Proc. of the Eleventh Intl. Symp. on Field Programmable Logic*, 2001.

[4] P. Diniz, M. Hall, J. Park, B. So, and H. Ziegler. Bridging the Gap between Compilation and Synthesis in the DEFACTO System. In *Proc. of the 14th Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*, Berlin, 2001. Springer Verlag.

[5] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of the Conf, on Design Automation and Test Europe (DATE99)*, March 1999.

[6] V. Kathail, S. Aditya, R. Schreiber, B. Rau, D. Cronquist, and M. Sivaraman. PICO: Automatically Designing Custom Computers. In *IEEE Computer*, pages 39–47, Sept. 2002.

[7] Mentor Graphics Inc. *Monet^TM User's Manual*, 2002.

[8] B. So, M. Hall, and P. Diniz. A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems. In *Proc. of the 2002 Conf. on Programming Languages Design and Implementation (PLDI'02).* ACM Press, June 2002.

[9] B. So, H. Ziegler, and M. Hall. A Compiler Support for Custom Data Layout. In *Proc. of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC'02)*, Berlin, 2002. Springer Verlag.

[10] Synopsys Inc. *Behavioral Compiler^TM. User's Guide, http://www.synopsys.com/products/beh_syn/beh_syn_br.html, 1999.*

[11] Synopsys Inc. *CoCentric^TM Data Sheet, http://www.synopsis.com/products/cocentric_studio/cocentric_studio.html, 2002.*

[12] M. Weinhardt. Compilation and pipeline synthesis for reconfigurable architectures. In *Proc. of the 1997 Reconfigurable Architecture Workshop (RAW'97).* Springer Verlag, 1997.