Cycle-time Aware Architecture Synthesis of Custom Hardware Accelerators

Mukund Sivaraman, Shail Aditya
Compiler and Architecture Research Laboratory
HP Laboratories Palo Alto
HPL-2002-300
October 21st , 2002*

E-mail: mukund@hpl.hp.com, aditya@hpl.hp.com

high-level synthesis, timing analysis, embedded hardware architecture synthesis

We present the cycle-time aware architecture synthesis methodology used in PICO-NPA that automatically synthesizes minimal cost RT-level designs from high-level specifications to meet a given cycle-time. This allows subsequent physical synthesis to succeed on first pass with predictable performance. The core of the methodology is a static timing analysis engine that is used at multiple levels - program-level, architecture-level and RT-level - in order to identify, schedule and validate useful operator chains that are incorporated into the design automatically. We present architecture synthesis results for several embedded applications and evaluate the benefits of this technique.

* Internal Accession Date Only Approved for External Publication
Presented at the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems -- CASES--8-11 October 2002, Grenoble, France

Cycle-time Aware Architecture Synthesis of Custom Hardware Accelerators

Mukund Sivaraman
Hewlett-Packard Labs
1501 Page Mill Road, MS 1166
Palo Alto, CA 94304
1-650-857-2549
mukund@hpl.hp.com

Shail Aditya
Hewlett-Packard Labs
1501 Page Mill Road, MS 1166
Palo Alto, CA 94304
1-650-857-3088
aditya@hpl.hp.com

ABSTRACT
We present the cycle-time aware architecture synthesis methodology used in PICO-NPA that automatically synthesizes minimal cost RT-level designs from high-level specifications to meet a given cycle-time. This allows subsequent physical synthesis to succeed on first pass with predictable performance. The core of the methodology is a static timing analysis engine that is used at multiple levels – program-level, architecture-level and RT-level – in order to identify, schedule and validate useful operator chains that are incorporated into the design automatically. We present architecture synthesis results for several embedded applications and evaluate the benefits of this technique.

Categories and Subject Descriptors
B.6.3 [**Logic Design**]: Design Aids – *automatic synthesis, optimization.*

General Terms
Design, Performance, Verification.

Keywords
operator chaining, high-level synthesis, timing analysis, delay analysis, timing during scheduling, target clock period, clock frequency, embedded hardware architecture synthesis.

1. INTRODUCTION
The ever increasing demand for multi-media consumer products, mobile devices, and other "smart" products in today's market poses an enormous design challenge to the embedded computer and electronics industry. The main objective is to turn out fast, reliable, and cost-effective electronic designs of increasing complexity in the shortest possible time. The PICO (Program-In-Chip-Out) project addresses this challenge by providing an automatic methodology to design programmable and non-programmable custom hardware accelerators starting from a high-level algorithmic description such as a C program. Starting from a high-level description enables comprehensive application analysis that is used for both high performance architecture design as well as reducing cost via customization. Furthermore, architectural design automation and exploration enables the user to make informed design choices leading to more cost-effective and reliable designs with a faster time-to-market.

This paper describes the cycle-time aware architecture synthesis methodology within PICO-NPA [1], the subsystem of PICO that designs custom, non-programmable loop accelerators automatically. PICO-NPA takes a C loop nest and a performance requirement as input and produces a customized hardware circuit of minimal cost that executes that loop at the desired performance level. The system is capable of making cost-performance trade-offs at multiple levels: exploiting multi-dimensional loop-level parallelism by spreading the computation over multiple processors (P), exploiting instruction-level parallelism within each processor through software pipelining as measured by the initiation interval (II) between successive loop iterations, and optimizing architecture-level hardware cost and performance by better utilization of the specified clock cycle-time (T). In this paper we focus our attention on this last issue.

PICO-NPA differs from traditional high-level synthesis techniques [2][3][4][5] in that it actively models and manages the achievable cycle-time of the circuit being designed to be close to a pre-specified cycle time T. Using the techniques described in this paper, PICO-NPA is able to automatically synthesize RT-level designs that are better suited for subsequent physical synthesis in that they achieve first pass timing convergence at the given cycle time with a high degree of confidence. At the same time, such designs are more cost-effective because of a better utilization of the available clock period through a systematic use of operator chaining during architecture synthesis.

The outline of this paper is as follows. Section 2 gives an overview of the PICO-NPA design flow. Section 3 discusses the timing-oriented view of the architecture synthesis problem in more detail. Section 4 describes our timing-driven architecture synthesis methodology, including ways to selectively reduce cost as well as to improve critical path timing. Sections 5 and 6 discuss our experimental setup and results respectively. Section 7 discusses related work and Section 8 concludes.

CASES 2002, October 8–11, 2002, Grenoble, France.

2. PICO-NPA DESIGN FLOW

The abstract design flow followed within PICO-NPA is shown in Figure 1. PICO-NPA accepts a multi-dimensional loop nest written in C that needs to be accelerated in hardware to a desired performance level, expressed as a combination of the number of processors (P), the initiation interval (II) and a cycle time (T). Architecture synthesis within PICO-NPA is actively driven by these parameters and has two major steps. In the first step, the given loop nest is analyzed for dependences and its iterations are mapped to an array of systolic processors. In the second step, the transformed loop structure is used to synthesize the architecture of a single systolic processor. The latter is achieved by first allocating sufficient hardware to meet the desired initiation interval and then scheduling the loop operations on that hardware in a software pipeline at the specified initiation interval and cycle-time.

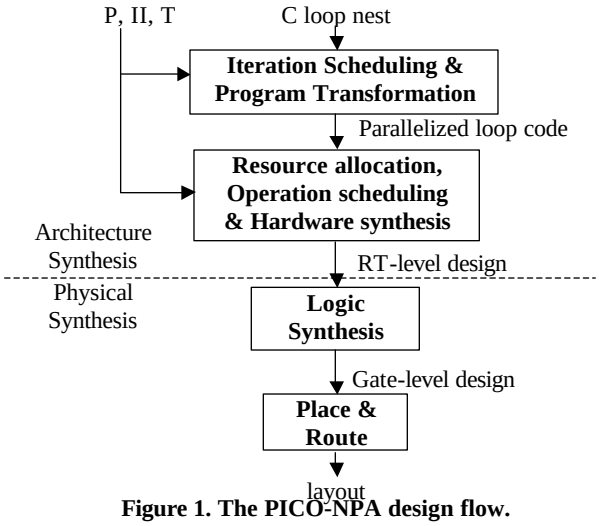


Figure 1. The PICO-NPA design flow.

The subsequent steps of physical synthesis (logic synthesis, place and route) are fairly standard and PICO-NPA uses standard commercial tools to implement them. However in traditional design flows, very often the design goes through several iterations of physical synthesis in order to meet the given cycle-time constraints through a tedious manual process of discovering critical paths in the design, fixing them in the RT-level design, and then re-synthesizing. PICO-NPA avoids such iterations by producing an RT-level design that is expected to meet its cost and performance estimates through subsequent physical synthesis in one pass. This is the main advantage of explicitly managing cycle-time constraints during architecture synthesis and is the main topic of this paper.

3. PROBLEM DESCRIPTION

Given a circuit, let us denote the maximum combinational delay that a signal may experience before being latched at a sequential latch by Δ. The metric (T-Δ) is called the available **timing slack** (or simply slack) at that latch. Usually, there is some variance in the amount of slack across all latches in the circuit. A typical histogram of the available slack at a latch is shown in Figure 2. The figure shows the number of latches that have a certain amount of available slack for a given cycle time of 10ns in the *channel* application.

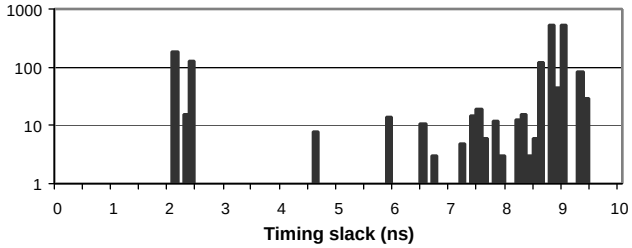


Figure 2. A histogram of available timing slack at T=10ns.

It can be seen that a majority of latches have a lot of slack and hence are "wasting" the computation time available, while others have relatively tighter signal arrival constraints. While it may not be possible (or desirable) to reduce the cycle-time of the circuit due to these tightly constrained latches, it is still possible to make use of the excess slack at other latches to reduce the overall cost of the circuit. This can be done by executing more than one flow-dependent operation in the same clock cycle and eliminating the intermediate latches between them. This technique is known as **operator chaining** [2] and is a well-known optimization in high-level synthesis. Operator chaining also potentially increases performance by reducing the total number of control steps needed to execute the program.

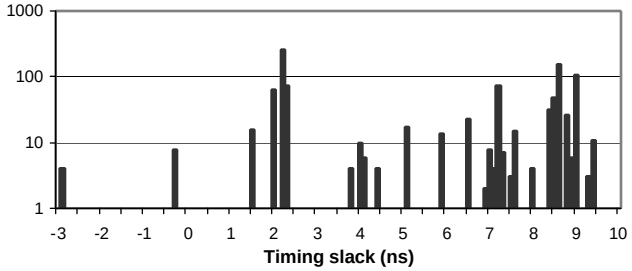


Figure 3. Slack histogram for unrestricted chaining at T=10ns.

Adding too much combinational logic to a control step, however, may cause the available slack to become negative, which implies that the circuit is unable to meet the specified cycle-time T and can only correctly operate at a reduced speed. This situation is depicted in Figure 3 for the *channel* application, where we have allowed unrestricted chaining in the circuit to take place that is limited only by the availability of resources and data dependences due to recurrences. This has led to a design that can only correctly operate with a cycle-time of 13ns.

A good RT-level design, therefore, is one where the available slack never goes into negative territory and is otherwise distributed as close to zero slack as possible in an attempt to reduce the overall cost of the design. A cycle-time aware design flow actively attempts to construct minimal cost designs that are guaranteed to

meet a specified cycle-time T as opposed to determining their maximal operating frequency after blind cost optimization.

In PICO-NPA, we tackle this problem during architecture synthesis (as opposed to physical synthesis in traditional design flows) because this provides maximal freedom in making intelligent operator chaining decisions. We use *global* information from the control and data flow graph (CDFG) prior to operation scheduling in order to identify what sequences of operations should be considered for chaining, and we use *local* timing analysis during scheduling and hardware mapping in order to validate the candidate operator chains. Our goal is to create the maximal chaining possible in the final design without impacting its cycle-time. Furthermore, we wish to achieve this even in situations where hardware resources may be shared between multiple operations (II>1).

4. APPROACH

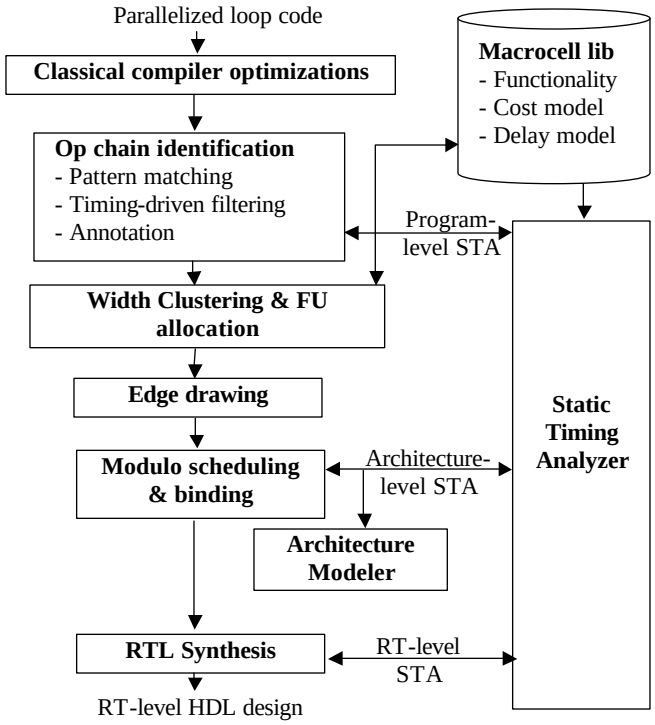


Figure 4. The cycle-time aware architecture synthesis design flow in PICO-NPA.

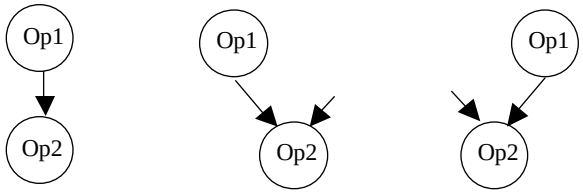
Figure 4 shows the cycle-time aware architecture synthesis design flow in PICO-NPA. The key feature of our approach is a **static timing analysis** sub-system that is used at multiple steps of the design flow to validate various architectural decisions being made with respect to cycle-time constraints. The design flow can be broadly classified into three phases based on the level of abstraction at which the timing analysis sub-system is used. In the pre-scheduling phase, *program-level* timing analysis is used to identify candidate operator chains. During scheduling and binding phase, *architecture-level* timing analysis is used to select specific operator chains to be incorporated into the design that meet the specified cycle-time. Finally, a detailed timing validation check is performed on the fully synthesized *RT-level* design in the post-scheduling synthesis phase.

4.1 Pre-scheduling

As shown in Figure 4, PICO-NPA starts by performing classical compiler optimizations on the parallel loop code obtained after iteration-level scheduling and mapping of the original C loop nest onto an array of systolic processors. The subsequent steps relevant to cycle-time aware architecture synthesis are described below.

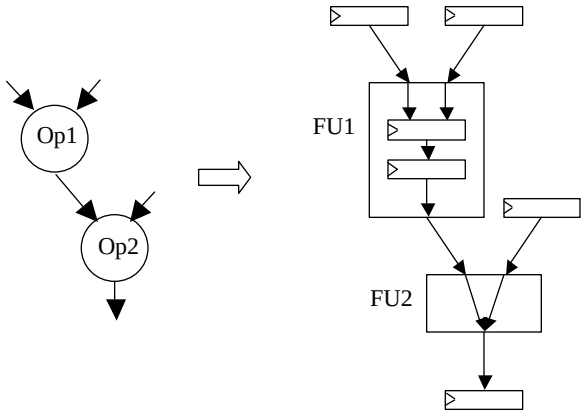
4.1.1 Operator chaining candidate identification

At this step, potential operator chains are identified using a pattern-matching engine. The pattern matcher reads in a library of templates specified in a general template specification language, and matches the templates with the operations and operands in the program graph. For the purposes of operator chaining, we use templates of the following form:



where Op1 can be an arithmetic, logical, comparison, or move operation and feeds into a unary operation Op2 (move, abs) or either operand of a non-unary operation Op2 (arithmetic, logical, or comparison). The templates can be used in effect to exclude certain operations from being chained due to system interface constraints. For example, loads/stores may be excluded if the path from/to external memory doesn't have enough timing slack. Note that these templates only specify potential operator chains of length equal to 2. Longer chains of operations may be formed by the concatenation of back-to-back operator chains during scheduling (Section 4.2). Therefore, these templates are sufficient for identifying all chaining opportunities including those across loop iterations.

For each pattern that is identified by the pattern matcher, a program-level timing analysis is performed as follows in order to check if the pattern is indeed chainable. As a first step, for the pattern under consideration, each operator in the pattern is mapped to its fastest hardware implementation (the one with maximum available slack) from among the macrocells available in the macrocell library. In the second step, we evaluate the delay of every latch-to-latch signal path induced by chaining the operations in the pattern using the components chosen above, and check against the specified cycle-time T. The mapping from an operator chaining pattern to corresponding hardware is illustrated below:



Here, Op1 is mapped to FU1, which happens to be internally pipelined, while Op2 is mapped to FU2, which is combinational. The signals going in and out of the pair of FUs are terminated in latches and the resulting hardware structure is checked for timing validity, as described later in Section 4.4.

Choosing the fastest hardware mapping results in an optimistic timing check, therefore, any pattern that fails to satisfy timing at this point will certainly not meet timing when the corresponding operations get scheduled and bound onto actual FUs selected. Such timing-violating patterns are pruned away, thereby reducing the total number of potential chains that the scheduler needs to consider.

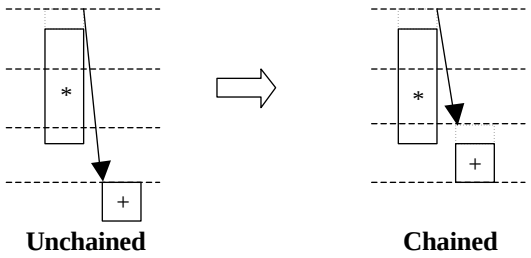
For those patterns that pass the timing check, a chaining attribute is annotated on the program graph's internal representation (IR). These attributes are kept up to date as further compiler optimizations take place, and are used in a subsequent edge drawing step (Section 4.1.3).

4.1.2 Width clustering and FU allocation

Operations are grouped into clusters based on their width [8], and for each width cluster, a least-cost set of FUs are allocated from a library of macrocells using a mixed integer-linear program (MILP) formulation [1]. In the course of formulating this MILP problem, we determine whether a given FU can execute a given operation type. In addition, we also ensure that the delay of the FU when executing the widest operation in the width cluster of this type satisfies the specified clock cycle-time. For instance, a multiply operation executing on a pipelined multiplier FU may meet the specified cycle-time, but may not do so on a combinational multiplier FU. Therefore, the combinational multiplier would be excluded from consideration for this multiply operation during FU allocation.

4.1.3 Edge drawing

Dependence edges are drawn for the program graph prior to scheduling. At this point, the flow-edge latencies[1] between potentially chainable operations are reduced by 1. For instance, if a 3-cycle multiply operation can be chained with an add operation, then the flow-edge latency from the multiply operation to the add operation will be marked as 2 cycles instead of 3 cycles, as illustrated below:



This allows us to effectively convey potential operator chaining information to our scheduler which works with integer latencies.

4.2 Scheduling and Binding

We have extended the iterative modulo scheduler (IMS [6]) to make scheduling and binding decisions that are correct with respect to timing, i.e., they do not lead to violation of the cycle-time. At each step, the scheduler picks up the highest priority operation and tries to schedule it at the earliest time slot at which an FU capable of executing that operation is available. The combination of a specific time slot and a specific FU resource is called a *scheduling pattern*. We actively manage timing slacks during this phase by checking the timing validity of every scheduling pattern that the scheduler considers for an operation. This involves ensuring that selecting this pattern will result in hardware where every path satisfies the given clock cycle-time constraint.

The validation of a scheduling pattern crucially depends on previous scheduling and binding decisions because they directly determine the physical connectivity among the hardware FU components. Therefore, we use an **architecture modeler** that maintains an internal representation of the partial hardware structure as it is being defined during the scheduling process. The architecture modeling can be fine-grained wherein data flow between FUs is bound to registers and the corresponding interconnect is synthesized, or can be coarse-grained wherein data flows are modeled as virtual links between producing and consuming FUs. In our observation, the coarse-grained architecture modeling proved to be sufficiently accurate for timing analysis purposes.

In order to perform the timing validation, the architecture modeler temporarily updates the hardware structure with the scheduling pattern under consideration. Architecture-level timing analysis is then performed as discussed below to check if this resulting hardware structure can be clocked at the specified cycle-time. If the scheduling pattern fails the timing validity check, then it is removed from consideration at this scheduling step.

Both the hardware structure updation and timing analysis occur in the innermost loop of the modulo scheduler, therefore, they need to be very efficient. We perform these steps incrementally, i.e., at

[1] The flow-/anti-/output-edge latency between two operations corresponds to the minimum operation issue time separation such that the flow-/anti-/output-dependence relations are satisfied [7].

each step, timing analysis is performed only for those portions of the updated hardware structure whose timing is affected. An example of this is shown in Figure 5. It shows the addition of virtual data flow links (shown as bold arrows) to the hardware structure as a result of scheduling and binding an operation on FU1, and the portion of the updated hardware structure where timing analysis would need to be performed (the shaded region).

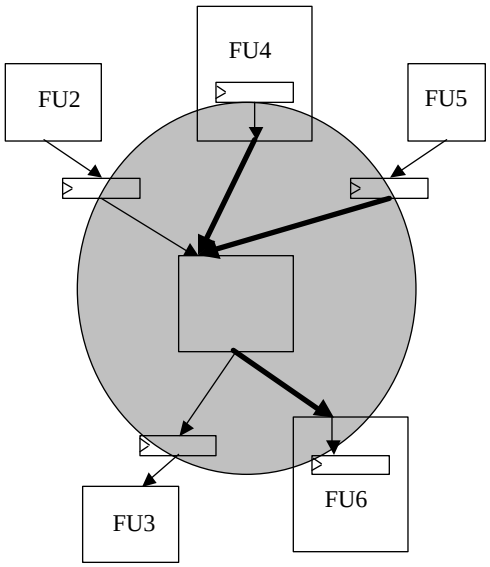


Figure 5. Incremental hardware updation and timing-validity checking.

We make certain conservative approximations during the timing analysis in order to avoid a situation where the current scheduling and binding decision becomes timing-invalid in the future as more operations are bound on an FU (thereby causing its width, fanout load capacitance, or fanin operand multiplexing to increase). The width, fanout and fanin of each FU is approximated prior to scheduling and binding, based on the cluster to which the FU belongs, the maximum fanout over all operations in the program graph, and the II.

4.3 Post-Scheduling RT-level Timing

After scheduling & binding of operations, binding of variables to registers and subsequent materialization of the hardware has been performed, timing analysis is done on the detailed RTL hardware structure. At this point, the FU widths, fanouts, register and interconnect structure are known exactly, therefore, the timing analysis is very accurate. This timing analysis serves as validation of the design for timing correctness.

4.4 Timing analysis

As described above, the timing analysis sub-system of PICO-NPA is used at various steps in the design flow at different levels of abstraction. However, the underlying delay models and the timing analysis algorithm are common to and are shared across all of them.

Every RT-level hardware component (FUs, latches, switching multiplexers and logic elements) used in PICO-NPA has a delay model associated with it, which is included as part of a macrocell library database. For instance, the delay model for a 3-cycle pipelined multiply-adder (which reads its add input operand *in2* two cycles after operands *in0* and *in1* are read) is shown in Figure 6. The delay model consists of a set of timing edges from input ports or internal pipelining latches of a hardware component to other internal pipelining latches or output ports, with associated delay functions (Δ_{in}, Δ_{pipe}, Δ_{out}, Δ_{thru}). The delays are a function of relevant hardware parameters, e.g., FU width, output load capacitance etc. The delay functions may be taken from the datasheet for a hard-macro, or they may be derived by performing logic synthesis for several combinations of the relevant parameters and subsequently measuring the delays. Furthermore, these delays may be represented as closed-form functions, or as a set of values upon which interpolation may be performed as needed.

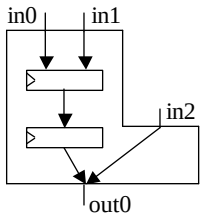


Figure 6. Delay model for a multiply-adder.

The timing analysis algorithm operates on a directed graph built using the delay models of a given set of hardware components and edges representing the physical connectivity between their output and input ports. Unconnected input and output ports are terminated by a latch, as shown in Figure 7. The algorithm finds the maximum arrival time of signals at each node by recursively finding the maximum signal arrival time at all its predecessor nodes, which is a linear-time algorithm [9]. In Figure 7, each timing edge is shown annotated with a delay value (interconnect delay is assumed to be zero for illustration purposes), and the computation of the maximum signal arrival time (*maxdelay*) at each node is also shown.

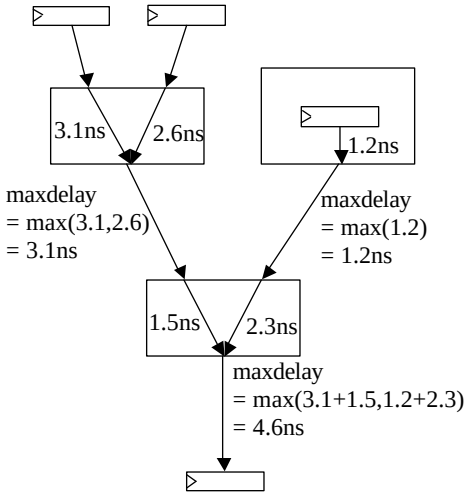
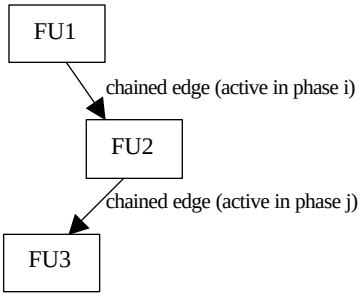
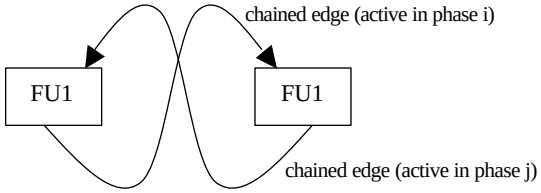


Figure 7. Timing analysis on hardware structure.

In our timing analysis, we also exploit the fact that the hardware has a periodicity of II cycles to eliminate false paths and combinational cycles from the analysis. A path is said to be false or *unsensitizable* when a signal cannot propagate from the beginning to the end of the path under any combination of actual inputs. An example of such a case is shown in Figure 8a. Here the two chains are not active in the same phase, therefore any path through all three FUs is not sensitizable. False combinational cycles may arise when two different operator chains share FUs, but in different directions, as shown in Figure 8b.



(a) An acyclic false path.



(b) A cyclic false path.

Figure 8. Examples of false paths.

Both acyclic and cyclic false paths are dealt with by performing the timing analysis separately for each of the II phases of circuit operation, where, for each phase, the timing analysis considers only those timing edges that are active in that phase.

5. EXPERIMENTAL SETUP

The cycle-time aware architecture synthesis framework described above has been implemented within our PICO-NPA research prototype. We have taken several applications through architecture synthesis and a few of them all the way through physical synthesis. The applications are taken from various domains such as printing, digital photography, signal processing and communications. Table 1 presents a brief description of the applications used.

Table 1. The embedded applications used in this study.

Application	Description
Cell	ATM cell delineation
Channel	Multi-channel ATM cell de-mux
Fir	16-tap finite impulse response filter
Fsed	Floyd-Steinberg image half-toning
Linescreen	Table-driven image half-toning
Heat	1D relaxation
Dct	8x8 Discrete cosine transform
Huffman	Huffman encoding
Matmul	Matrix multiplication
Sobel	Image edge detection
String	String sequence matching

For each application, we synthesized a single processor (P=1) running at a given clock frequency of 100MHz (T=10ns) for a range of throughput requirements (II=1, 2, 4, 8). We performed architecture synthesis in three different modes – one without chaining, one with maximal chaining without regard to timing slack[2], and one where chaining is performed in conjunction with active timing slack management. The timing slack and gate count numbers shown below were obtained by analyzing the RT-level netlist produced by PICO-NPA after architecture synthesis. The area and delay models used for the macrocells were derived by pre-characterizing the macrocells using logic synthesis, and curve-fitting the results to produce closed-form delay functions.

6. RESULTS

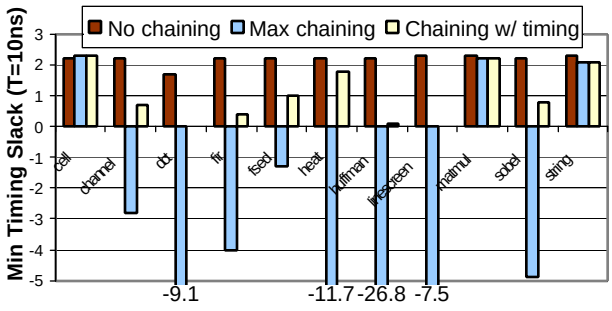


Figure 9. Minimum timing slack for II=1.

Figure 9 shows the results of timing slack management for II=1. On the Y-axis we have plotted the minimum slack across all latches in the circuit. A negative minimum slack implies that the

[2] This was accomplished simply by switching off the timing sub-system and accepting every chaining decision as being valid for timing.

circuit is unable to meet the specified cycle-time. Figure 9 clearly shows that chaining without timing slack management leads to circuits that cannot meet their specified cycle-time. It also shows that timing slack management is very effective in controlling the amount and location of operator chaining so that the given cycle-time is met across all paths. Without this mechanism, for many applications the only way to meet the given cycle time is to do no chaining at all. We also see that some applications remain unaffected, either because there are very few opportunities to chain (e.g. matmul) or all the chaining opportunities already meet the 10ns cycle time (e.g. cell and string). Applications in the latter category may still need active slack management for a faster clock design.

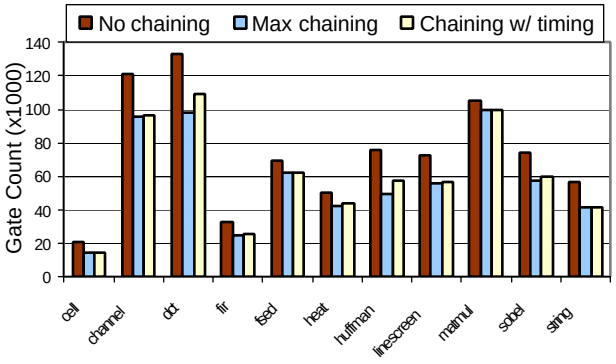


Figure 10. Gate counts for II=1.

The main benefit of doing operator chaining is a reduction in silicon area used by the circuit as measured by its gate count. The results for II=1 are shown in Figure 10. In general, the gate counts reduce when chaining is employed due to elimination of pipeline latches between the producer and the consumer function units. The chart shows that the potential reduction in total gate cost with maximal chaining varies dramatically among applications, ranging from 5% in matmul which provides very little opportunity to about 34% in Huffman where almost a third of the cost can be eliminated. However, the key observation is that chaining with timing slack management is also able to obtain most of this cost benefit ranging from about 5% in matmul to about 30% in cell. With timing slack management, the cost benefit is reduced in precisely those applications that would have otherwise performed over-aggressive chaining and failed to meet their cycle-time. This shows that our active timing slack management scheme is very effective in trading off a slight increase in silicon area in order to obtain a predictable cycle-time.

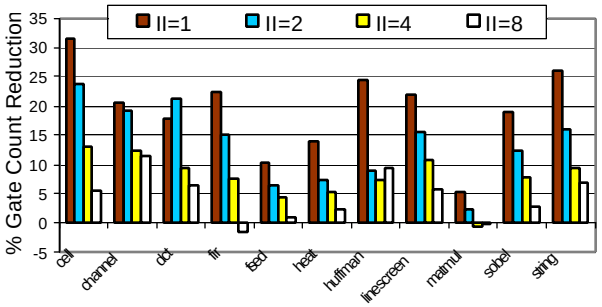


Figure 11. Percentage gate count reduction across II.

Figure 11 shows the percentage of gate count reduction obtained by chaining using timing slack management at lower throughput[3] (II>1). The main trend in the chart is that the cost benefit is lowered at higher II. This is as expected because at lower throughput, more and more hardware components (function units and latches) are shared and cost benefits can be realized only when all data-dependent operation pairs between two function units are actually chained. However, the encouraging aspect is that the degradation is gradual, and the cost reduction is still significant for several applications at II=8 (up to 10%). However, for some applications the cost even increases slightly (fir at II=8). This is attributable to the fact that chaining causes the schedule to change and therefore may adversely affect hardware resource sharing obtained previously. The interaction between the timing and resource management heuristics during scheduling needs further investigation and is beyond the scope of this paper.

As a final point, we expect the reduction in latches obtained by our approach to be retained by logic synthesis because commercial logic synthesis tools do not change the state elements specified in the RT-level design (retiming is rarely used). Furthermore, latch elimination will result in exposing more combinational logic between successive latches to logic synthesis, thereby leading to more opportunities for logic optimization, and consequently even larger gate count reductions than shown by our RT-level results.

7. RELATED WORK

Operator chaining, by itself, is a well-known technique in high-level synthesis. Various high-level synthesis systems either directly or indirectly address chaining within their framework [3][4][5]. Some systems make explicit chaining decisions either globally (e.g. multiply-add) or while scheduling operators from the CDFG into control steps using an abstract model of operator delays. Others are based on retiming [10] the final circuit after a preliminary round of scheduling and mapping and readjusting the placement of latches to reduce cost. In either case, the maximum speed at which the circuit may operate is an *output* of the design process, to be determined and improved upon during physical synthesis.

Our approach, in contrast, actively shepherds the architecture of the circuit to meet a cycle-time given as *input* using both global information before scheduling and local information during scheduling and hardware mapping. The designs produced by these techniques meet the specified cycle-time in a single pass of physical synthesis. Snider's work [11] is closest to ours in this respect but he does not consider II>1 designs (and the resulting complexity due to hardware resource sharing) and therefore his architecture-level timing analysis is highly simplistic.

8. SUMMARY AND CONCLUSIONS

In this paper, we have presented a description and evaluation of the cycle-time aware architecture synthesis methodology used within PICO-NPA. We use a multi-phase strategy that identifies useful operator chains at the program-level prior to operation

[3] For some experiments (II=2 : cell, dct; II=4 : heat, huffman), the modulo scheduler succeeded in scheduling only for target II + 1. This was due to the heuristic nature of our scheduling algorithm, but it does not affect our conclusions in any significant way.

scheduling, implements a subset of them during scheduling using active timing slack management at the architecture-level, and finally verifies these decisions after synthesis at the RT-level.

The major contributions of this paper are the following:

- We have shown that timing slack management during architecture synthesis is an effective technique in designing circuits automatically that can meet a specified cycle-time.
- We have shown that operator chaining is very effective in significantly reducing the cost of custom hardware accelerators generated by PICO. However, unrestricted use of operator chaining makes the achievable cycle-time of the circuit highly unpredictable. The active timing slack management framework described in this paper is able to make intelligent chaining decisions that provide most of the cost benefit without impacting the cycle-time.
- The cost reduction due to chaining is maximum for high throughput designs (5-30% for II=1). This improvement is lower but still fairly significant (up to 10% for II=8) for designs with lower throughput.

9. ACKNOWLEDGEMENTS

The authors would like to acknowledge Scott Mahlke and Shivarama Rao Kokrady for the initial implementation of the pattern-matching engine, and Mike Schlansker and Rodric M. Rabbah for an initial implementation of the architecture modeler.

10. REFERENCES

[1] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist and M. Sivaraman. PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators. *Journal of VLSI Signal Processing, 31, 2002,* pp. 127-142.

[2] R. Camposano. From Behavior to Structure: High-level Synthesis. *IEEE Design & Test of Computers*, October 1990.

[3] H. De Man, F. Catthoor, G. Goossens, J. Vanhoof, J. V. Meerbergen, S. Note, J. Huisken. Architecture-Driven Synthesis Techniques for VLSI Implementation of DSP Algorithms. *Proceedings of the IEEE*, 78(2), pgs. 319-335, February, 1990.

[4] N. Park and A. Parker. Sehwa: A software package for synthesis of pipelined data path from behavioral specification. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, March 1988, pp. 356-370.

[5] P. G. Paulin , J. P. Knight , E. F. Girczyc. HAL: A Multi-paradigm Approach to Automatic Datapath Synthesis. *Proc. Design Automation Conference*, 1986, pp. 263-270.

[6] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. *In Proc. 27th Annual International Symposium on Microarchitecture*, pgs. 63-74, December 1994.

[7] B. R. Rau, V. Kathail, S. Aditya. Machine-description driven compilers for EPIC and VLIW processors. *Design Automation for Embedded Systems*, Vol 4, pgs. 71-118, Kluwer Academic Publishers, Boston, 1999.

[8] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, T. Sherwood, Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol 20(11), November 2001.

[9] T. Kirkpatrick and N. Clark. PERT as an aid to logic design. *Tech. Rep. IBM*, 1966.

[10] C. E. Leiserson and J. B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6(1):5-35, 1991.

[11] G. Snider. Performance-Constrained Pipelining of Software Loops onto Reconfigurable Hardware. *Tenth ACM International Symposium on Field-Programmable Gate Arrays*, Feb. 2002.