Application Acceleration with the Explicitly Parallel Operations System – the EPOS Processor

Alexandros Papakonstantinou, Deming Chen, Wen-Mei Hwu
Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign
{apapako2, dchen, w-hwu} @uiuc.edu

***Abstract*- Different approaches have been proposed over the years for automatically transforming High-Level-Languages (HLL) descriptions of applications into custom hardware implementations. Most of these approaches however are confined by basic block level parallelism described within the CDFGs (Control-Data Flow Graphs). In this work we propose a new high-level synthesis flow which can leverage instruction-level parallelism (ILP) beyond the boundary of the basic blocks. We extract statistical parallelism from the applications through the use of Superblocks and Hyperblocks formed by advanced front-end compilation techniques. The output of the front-end compilation is then used in our high-level synthesis in order to map the application onto a new domain-specific architecture named EPOS (Explicitly Parallel Operations System). EPOS is a stylized micro-code driven processor equipped with novel architectural features that help take advantage of the instruction-level parallelism generated in the front-end compilation. A novel forwarding-path optimization engine is also employed during the high-level synthesis flow in order to minimize the long interconnection wires and the multiplexers in the processor. To evaluate the EPOS processor, we compare its performance with a previous domain-specific processor NISC on a common set of benchmarks. Experimental results show that significant performance gain (3.45X on average) is obtained compared to NISC.**

I. Introduction

As the deep sub-micron technologies continue to increase the per-die transistor count, High Level Synthesis (HLS) has regained a lot of attention during the last decade in the engineering community. One of its main advantages is the higher productivity through the use of higher level abstractions. This helps to bridge the growing gap between chip capacity and its efficient utilization by design engineers. There have been extensive research efforts on this topic [4,17,18]. Most of these efforts endeavor to transform the sequential high-level description into a system of parallel computing elements based on the CDFG (Control-Data-Flow-Graph) abstraction, where people work with basic blocks within the CDFG. In this work, we explore a new high level synthesis flow which can leverage instruction-level parallelism (ILP) beyond the boundary of basic blocks. Our experiments show that applications can be synthesized with significantly improved performance results over basic-block level HLS tools.

In our synthesis flow, statistical parallelism is extracted from the applications through the use of *superblocks* [14] and *hyperblocks* [11] that are formed by advance front-end compilation techniques within the IMPACT compiler [15]. The output of the compiler is then used in our HLS flow in order to map the extracted parallelism in a highly efficient way onto a domain-specific processor, named EPOS (Explicitly Parallel Operations System). EPOS is a stylized, microcode-driven processor equipped with novel architectural features that can take advantage of the maximum instruction-level parallelism extracted in the front-end compilation. Apart from optimizing the execution latency through instruction-level parallelism handling, our HLS tool also focuses on the clock frequency optimization by using a forwarding-path optimization engine. The goal of this engine is to bind operation onto the allocated resources so as to minimize the number of the forwarding (FW) paths and the multiplexers involved in the forwarding network of EPOS.

Fig. 1 gives an outline of our synthesis flow. Initially, we leverage the advanced compiler optimizations available in IMPACT to transform the original C code into *Lcode*, a three-address intermediate representation. Lcode is optimized through traditional compilation techniques and advance ILP extraction techniques that use profiling to generate superblocks and hyperblocks. Lcode is then fed to our scheduler together with the user-specified resource constraints, in order to produce scheduled Lcode. This Lcode is not yet bound to the functional units of the processor. Binding is done during the last step of the flow, during which the data forwardings entailed in the scheduled Lcode are considered. A min-cost flow algorithm is used to bind the operations on the FUs, while minimizing the FW paths and the corresponding operand multiplexing.

The development of the EPOS architecture offers several advantages. First of all, the control of the datapath through the micro-code words stored in a micro-code memory (more details in section 3) allows a flexible and simple alternative compared to i) memory-stored instructions, which require complex decoding logic and ii) complicated Finite State

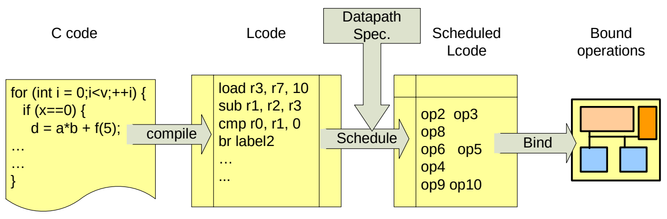


Fig. 1. High-level synthesis flow

Machines (FSM) which limit the size of applications that can be efficiently synthesized. Secondly, the instruction-level parallelism can easily be exploited by mapping the scheduling output onto micro-words. Finally, the use of the micro-code memory allows the customization and use of the EPOS processor for a domain of applications, rather than a single application. Thus, it trades off flexibility and performance, providing an ideal solution between a general purpose processor and an application-specific processor.

In the next section we first review related research for synthesizing applications described in high-level languages. Then in section 3 we describe the EPOS architecture in more detail. Section 4 contains information on the formulation and implementation of our scheduling and binding techniques, and section 5 provides experimental results. Section 6 concludes the paper and lists directions for future work.

II. Related Work

While many research results have been published in the HLS area [4,17,18], we will concentrate on several major related works. J. Babb et. al [7] focused on extracting parallelism by splitting an application into tiles of computation and data storage with inter-tile communications based on virtual wires. Virtual wires comprise the pipelined connections between endpoints of a wire connecting two tiles. Application data is distributed into small tile memory blocks and computation is then assigned to the different tiles. This work can produce efficient parallel processing units for the class of applications that can be efficiently distributed into equal data and computation chunks. However, applications with control intensive algorithms could result in contention on the communication through the virtual wires, imposing many idle cycles on the distributed datapaths.

In a different approach, S. Gupta et. al. [12,13] have focused on extracting parallelism by performing different types of code motions and compiler optimizations in the CDFG of the program. In particular, they maintain a Hierarchical-Task-Graph (HTG) besides the traditional CDFG. The nodes in an HTG represent HLL constructs, such as loops and if-then-else constructs. The authors show that their heuristics (named SPARK) offer significant reductions both in the number of controller states and also in the latency of the application. However, SPARK flow imposes limitations on the C features that it supports. Moreover, all the code motions are done on the initial CDFG with basic blocks, which may limit the amount of optimizations.

Another approach targeted an instruction-less architecture, NISC, proposed by M. Reshadi et. al [8]. This custom processor architecture removes the abstraction of the instruction set and compiles HLL applications directly onto a customizable datapath which is controlled by either a memory of control words, or a traditional FSM circuit. The main benefits of this approach are the reprogrammable and decode-free features. The compilation of the NISC system is based on a concurrent scheduling and binding scheme on basic blocks. Our processor architecture, EPOS, builds on this instruction-less architecture by adding new architectural elements and employing novel scheduling and binding schemes for exploiting instruction-level parallelism beyond basic blocks.

The increasingly significant effect of long interconnects on power, timing and area has led to the development of interconnect-driven HLS techniques. J. Cong et al. [19] have looked into the interconnect-aware binding of a scheduled DFG on a distributed register file microarchitecture (DRFM). Based on the same DRFM architecture, K. Lim et al. [20] have proposed a complete scheduling and binding solution which considers minimization of interconnections between register files and FUs. EPOS on the other hand, uses a unified register-file (RF) and allows results to be forwarded directly from the producing to the consuming FUs for reduced latency.

III. Epos

A. *The EPOS Philosophy*

Extracting instruction-level parallelism can be done either statically [10] (at compile time) or dynamically [16] (at execution time). Dynamic extraction of parallelism is based on complex hardware, while static techniques [2,6] shift the burden of identifying parallelism onto the compiler [1,5]. Thus, extracting ILP statically simplifies the hardware and enables higher clock frequencies. This strategy is known as the EPIC (Explicitly Parallel Instruction Computer) [9] philosophy. EPOS is based on this philosophy. The custom processor follows the execution plan created by the synergy of the compiler and the scheduling and binding engine. Special architectural elements are added to the main datapath architecture to handle potential mis-predictions of the static parallelism extraction. Since the custom processor is compiled for a specific application or a domain of applications, static parallelism extraction can offer significant performance benefits with a minimal hardware cost.

B. *Front-End Compilation*

For the extraction of the statistical ILP from the application, we use the IMPACT compiler, which transforms the HLL into Lcode. Lcode goes through various classic compiler optimizations and also gets profiled. Then the profile information is used to create superblocks and hyperblocks. Superblocks are formed by selecting frequently executed control paths in the program that span many basic blocks and grouping them into a single superblock that may have multiple side exits but only one entry point at the head of the block. Hyperblocks on the other hand, differ from superblocks in the way they select which basic blocks to merge in a single block. In particular, hyperblocks may group basic blocks that are executed in exclusive paths in the original program flow by predicating the instructions. Instruction predicates are stored in a predicate register-file. Hyperblocks, like superblocks, may have multiple side exits and a single entry point.

C. *EPOS Architecture*

The main elements of the EPOS architecture are the micro-code memory banks, which store the plan of execution as determined by the compiler and the scheduler. Each micro-code word contains control bits that determine the flow of data in the processor datapath for a single cycle. Each micro-word can be split into multiple memory banks that are potentially placed close to the datapath elements they control, thus facilitating better routing. There is also a micro-word address (MWA) register that holds the current micro-word to be executed, while micro-word address generation logic chooses the next micro-word address. The functional units can have different characteristics in terms of their latency, pipeline and functionality characteristics.

As shown in fig. 2 there are two RFs, one for program values and one for predicate values (PRF), and each FU may store its results into a small shifting register file (SRF). Each FU result is stored in the top register of its respective SRF while previous values are shifted one position lower in the SRF. This allows for predicated operations to be speculated or in other words promoted over the predicate definition operation by a few cycles. Thus, promotion can be achieved without the need of a unified shadow RF. Simple circuitry is used to squash the false predicated operations while allowing the correct operations to store their results in the RF. Forwarding paths from the outputs to the inputs of FUs and register-file bypassing (RFB) are used to optimize the performance of data-intensive applications. For a result produced in cycle *n*, forwarding allows its use in cycle *n+1*, while RFB allows its use in cycles *n+2* and *n+3* (assuming 2-cycle RF writes). In this architecture we use the SRFs for both register-file bypassing and predicated operation speculation.

IV. SCHEDULING & BINDING

The scheduling engine of our synthesis flow tries to extract the maximum parallelism possible out of the speculated and predicated Lcode, so as to reduce latency under resource constraints. Binding, on the other hand, has the capability to reduce the clock period of the design through the reduction of interconnections and multiplexers.

A. *Scheduling Engine*

The algorithm used for scheduling the Lcode operations is a variation of the original list scheduling algorithm [4]. This algorithm is designed to handle FUs of different latency and pipeline characteristics as well as the intricacies of predicated

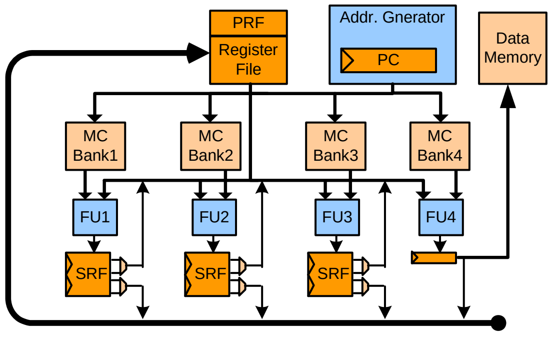


Fig. 2. EPOS architecture overview

and speculated operations.

Initially, a direct acyclic graph (DAG), $G_d = (V, A)$, is built based on the dependence relations of the Lcode operations. Set V corresponds to Lcode operations and set A corresponds to 3 different types of dependence relations between the operations: data dependences (Read-after-write), predicate dependences and flow dependences

The data dependence arcs represent real dependences between producer and consumer operations. The dependences of predicated operations on predicate defining instructions are represented with the predicate dependence arcs. This differentiation between predicate and data dependences is done in order to handle speculation of predicated operations with the hardware support of the Shift-Register-Files. Finally, the flow dependence arcs are used to ensure that branch and store operations are executed in their original order within Lcode, i.e. avoid speculation of such operations. Mis-speculation of these types of operations may lead to incorrect execution and requires complex hardware to fix.

After the data dependence graph construction, global and local slack values are computed for each node of the graph. Global slacks represent an operation's criticality in the dataflow of an entire function, while local slacks represent the operation's criticality within its block. Local and Global Slacks are used in a weighted function to determine the total Slack. For example, assume the following operation sequence within a superblock: op1$\rightarrow$br$\rightarrow$op2, where a side branch (br) is between two operations. If both operations have high local slacks then the global slack will determine which one has priority to be scheduled. If op1 has a relatively low global slack due to the control path that goes along the branch, then it will be given priority to optimize the schedule for the less likely case (i.e. the side exit off the hyperblock/superblock). On the other hand, local slack is computed within the boundaries of a superblock/hyperblock and thus promotes the operations with low slack in the most likely case (i.e. the path through the superblock/hyperblock).

Subsequently, list-scheduling is performed on a per-block basis taking into consideration the different types of dependences. For example data dependent operations can not enter the ready list until their data producer is scheduled and finished executing. On the other hand predicate dependent operation in a system with SRFs can be scheduled a number of cycles, equal to the SRF depth, ahead of their predicate producer. Flow dependences are also not as strict dependences as data dependences. That is, a flow dependent operation can be scheduled before the operation it is dependent to has finished execution. For example a store operation can be scheduled in the same cycle with a branch that it is flow-dependent to. If the branch turns out to be taken, the squashing logic (used for false predicated instructions) can terminate the store operation before it updates the memory

B. *Binding Engine*

First let us define *forwarding path* (FWP) to signify the physical forwarding bus from the output of a FU to the input of another. Then we can define *data forwarding* (DFW) as the data value forwarded from one operation that ends in cycle n to another operation that starts in cycle n+1. Our objective in this phase is to bind the data forwardings to forwarding paths (through binding operations to FUs), so as to minimize the number of FWPs. Thus, we can reduce the number of long wires and the degree of multiplexing required while still honoring the schedule generated in the previous step. This will help reduce the critical path delays, which are often linked to the FWPs and will enable faster clocking of the processor. In order to achieve this goal we transform our binding problem into a clique partitioning one and then use a network flow formulation to solve the clique partitioning. A post-processing phase may be required to make the network solution feasible for our schedule.

B.1 Compatibility Graph

We use a modified version, $G_{d2} = (V, A_2)$, of the DAG constructed in the scheduling step, where set A_2 corresponds to the data dependences only. A new DAG, $G_{d3}=(V_3, A_3)$ is formed as shown in fig. 3b by pruning away the nodes that do not have any data flowing from/to operations in the preceding/next cycle of the schedule. The edges attached to the pruned nodes are also pruned away. Graph G_{d3} represents the forwardings entailed in the schedule, i.e. an edge $\alpha = (v_i, v_j)$ corresponds to a forwarded value from operation v_i to v_j. A compatibility graph $G_c = (V_c, A_c)$ for these forwardings (FWs) can then be constructed, as shown in Fig. 3c. V_c does not represent the operation nodes in G_{d3} but corresponds to all FWs (edges of G_{d3}) and a directed edge $\alpha_c = (v_m, v_n)$ is drawn between two vertices if the source operation of FW v_m is scheduled in an earlier cycle than the source operation of FW v_n. Each edge α_c is assigned a weight w_{mn}, which represents the cost of binding v_m and v_n to the same FWP.

Given a forwarding network represented by compatibility graph G_c, our goal is to find an edge subset in G_c that covers all the vertices in V_c in such a way that the sum of the edge weights is minimum with the constraint that all the vertices can be bound to no more than *k* FWPs, where *k* is the minimum number of FWPs required to fulfill the schedule. This can be translated into a clique partitioning problem, where each clique corresponds to the DFWs that can be bound into a single FWP.

B.2 Network Flow

We define *complex forwarding structure* (*CFS*) to represent the set of data forwardings that share the same source operation. CFS is needed to avoid infeasible forwarding bindings. For example in Fig. 3c, binding DFWs *f1* and *f7* to one FWP and DFWs *f2* and *f8* to a different FWP would be incorrect, since *f1* and *f2* have a common source and *f7* and *f8* don't. Combining them together into a CFS and treating it as a single forwarding pattern have effectively solved this problem. A new vertex *c* is added for each CFS in the graph. Vertex c is connected in such a way that the previous outgoing edges are unlinked and connected as outgoing edges to vertex c (Fig. 3d). The transformed compatibility graph is the basis for building network N_G = (s, t, V_n, A_n, C, K), where a source vertex *s* and a sink vertex *t* are added with edges linking them to all other vertices (dashed edges in Fig. 3d). N_G has the cost function C and the capacity K defined for each edge in A_n.

The cost, C, of the network edges tries to capture, among other things, the similarity of the neighboring forwarding patterns of two compatible FWs, so as to avoid infeasible solutions. The *Fschema* parameter is used to represent this factor and it is calculated based on G_{d3}. For example, let us consider the DAG shown in fig. 4a. The minimum number of FWPs required to satisfy this schedule is two. However, in order to produce a feasible binding with two FWPs, f1, f3 and f5 need to be bound to the same FWP while f2, f4 and f6 are bound to a second FWP. Otherwise, 3 FWPs will be required. This can be fulfilled with the help of the Fschema value. Fschema is calculated by trying to find the maximum match in the neighboring FW patterns. In fig. 4b, the bigger values produced for pairs f2-f4 and f3-f5 show that these pairs of forwardings have more similarities in their FW neighboring patterns. These values would help bias the network flow to find better solutions by binding similar pairs together.

The network graph is further modified by the split-node technique which ensures that each node is traversed by a single flow. This is achieved by splitting each node into two nodes connected with a directed edge of a single capacity (a similar technique is used in [3]). The problem of minimizing FWPs has been transformed into finding the min-cost flow in the network. When the min-cost flow is computed, k flows from the node s to the node t are produced. The nodes traversed by each flow should be bound to the same FWP.

B.3 Binding Solution Check and Post-Processing

By using the CFS concept and the Fschema values in the cost function we manage to avoid almost all the infeasibilities in the solution generated by the min-cost flow. However, some information regarding the relationship between the FWs is not visible in the compatibility graph, which may lead to

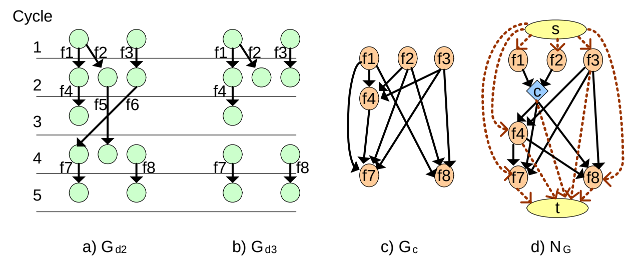
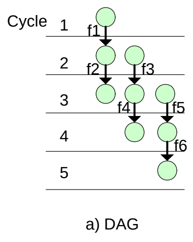


Fig. 3. Compatibility graph creation



	f2	f3	f4	f5	f6
f1	10	20	10	10	0
f2		10	30	0	10
f3			20	30	10
f4				10	20
f5					10

b) Fschema values

Fig. 4. Using Fschema values

infeasible bindings. For example, in fig. 5a two cliques have been formed: *clq1*={f1,f4,f5} and *clq2*={f2,f3}. This binding, however, is infeasible and can not be implemented with 2 FWPs. This becomes evident if we notice in fig 5a that the source FU of f5 is the same as destination FU of f4. However, according to the generated solution in fig. 5b clique clq1 that f4 and f5 belong to, has different destination and source FUs.

In order to handle such inconsistencies, min-cost flow is followed by a post-fix phase. During this phase the min-cost solution is translated into a set of rules that describe the relations between the FWPs. This set of rules is built by referencing the relation of the FWs in the initial DAG. For example, from FWs f2 and f3 in the DAG of fig. 5a and the min-cost solution described we can infer the rule: “Destination of clq2 is the same as Source of clq2”. Thus, the FWP corresponding to clq2 in fig 5b forwards values to itself. The full set of rules that are inferred for our example is as follows:

(1) Src(clq1) = Src(clq2) ; (2) Snk(clq1) ≠ Snk(clq2) ;
(3) Src(clq2) = Snk(clq2) ; (4) Src(clq1) ≠ Snk(clq1) ;
(5) Src(clq1) = Snk(clq2) ; (6) Src(clq1) = Snk(clq1) ;

where Src() and Snk() represent the source and the sink FUs of the FWP that the clique corresponds to. The infeasibility described before can now be formally discovered by searching for rules that contradict each other. In our example, rules 4 and 6 are incompatible. When two rules conflict, the rule with the least number of associated FWs is marked as conflicting and its associated FWs are un-bound. Subsequently the FWs associated with conflicting rules are either bound on alternative allocated FWPs (if feasible) or bound to newly allocated FWPs. In our example f5 is bound to a new FWP as shown in fig. 5c and a feasible solution with 3 FWPs is found.

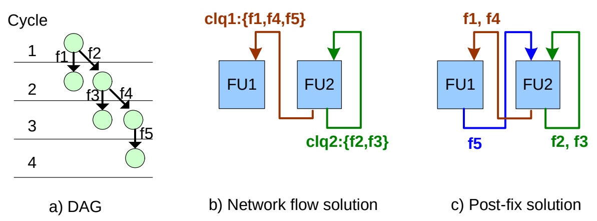
V. Experimental Results

A. *Performance Comparison to NISC*

To evaluate our HLS flow and EPOS architecture, we use a set of benchmarks and compare the performance results of EPOS with NISC.

A.1 Execution Cycles

First we focus on the number of cycles required for the execution of the application when synthesized by EPOS and NISC. Since NISC does not have register file bypassing and operation predication features, we turned these two features off in EPOS as well for this comparison, so as to measure the effect of our ILP-extracting synthesis flow. The datapath configuration used for all the experiments consists of 4 ALUs



a) DAG b) Network flow solution c) Post-fix solution

Fig. 5. Binding post-fix example

that execute arithmetic, logic and shifting operations, 1 Multiplier, and 1 LD/ST unit.

In table I we can see that there is a significant decrease in execution cycles for almost all benchmarks when they are synthesized on EPOS. The speedup gained in EPOS ranges from 1.07 to 6.85, with an average speedup of 3.13.

TABLE I
Clock - Cycles Comparison

Benchmark	*NISC*	*EPOS*	*Speedup*
mdct	146	61	2.39
bdist2	2110	1966	1.07
startup	2838	1464	1.94
bubble	31247	4563	6.85
dct	4920	2333	2.11
dijkstra	104610	23640	4.43
Average			3.13

TABLE II
Frequency Comparison (MHz)

	UnOpt. EPOS	NISC	Optimized EPOS		
				v.s. UnOpt.	v.s. NISC
mdct		110.00	103.37	+28%	-6%
bdist2		81.52	104.98	+30%	+29%
startup		118.20	113.58	+41%	-4%
bubble	80.76	103.33	113.68	+41%	+10%
dct		85.60	97.79	+21%	+14%
dijkstra		96.79	114.85	+34%	+19%
Average				+33%	+10%

TABLE III
MUX Comparison (Total MUX Inputs)

	UnOpt EPOS	NISC	Optimized EPOS		
				v.s. UnOpt.	v.s. NISC
mdct		54	60	-56%	+11%
bdist2		82	56	-59%	-32%
startup		46	52	-62%	+13%
bubble-sort	136	48	52	-62%	+8%
dct		76	64	-53%	-16%
dijkstra		54	64	-53%	+19%
Average				-58%	+1%

A.2 Clock Frequency

In order to evaluate our forwarding path binding technique we compare the critical paths of the synthesized processors. The data and control memories are stripped off in both NISC and EPOS and only the datapath, the register file and the FWPs with the multiplexers are synthesized. Synthesis and timing analysis were done in Altera’s Quartus II environment. First, we built EPOS with all the possible FWPs (i.e. without any FWP optimization). Then, we performed the binding optimization to optimize FWPs. The results are listed in tables II and III. The second column “UnOpt EPOS” shows the results for the unoptimized EPOS, i.e the EPOS with a full set of FWPs. Table II lists the reported frequencies and table III shows the total size of the multiplexers (i.e. the total number of mux inputs). We can see that there is a correlation between the frequency and the MUX size. The binding optimization of EPOS minimizes the number of FWPs which has a large impact on the total required size of multiplexing and consequently on the critical path delays. We can observe that compared to unoptimized EPOS, the optimized EPOS reports up to 41% improvement on frequency and up to 62% reduction on total MUX size. Compared to NISC, EPOS achieves higher clock frequencies in most cases, while the MUX size is on average the same. By combining the execution cycles with the achieved frequency for both processors we can compare the benchmark execution latencies. These results are shown in fig. 6 and an average speedup of 3.45X over NISC is observed.

B. *Boosting Performance Further on EPOS*

The number of execution cycles on EPOS can be further improved by either turning RFB on or activating predicated operation speculation (POS). Table IV shows the speedup that can be achieved. The 2nd column lists the cycle numbers without RFB and POS. Columns 3 and 4 show the cycle number and speedup correspondingly when RFB is enabled, and columns 5 and 6 show the cycle number and speedup (over the 1st column figures) when both RFB and POS are enabled. With these architectural features we can further reduce the cycle numbers by up to 40% compared with the simple EPOS.

TABLE IV
EPOS SPEEDUP WITH RFB AND POS

	EPOS	*EPOS-RFbp*		*EPOS-RF-POS*	
Benchmark	cycles	cycles	speedup	cycles	speedup
mdct	61	57	1.07	57	1.07
bdist2	1966	1454	1.35	1390	1.41
startup	1464	1268	1.15	1088	1.35
bubble	4563	3989	1.14	3655	1.25
dct	2333	2045	1.14	2045	1.14
dijkstra	23640	20685	1.14	18212	1.30
Average			1.17		1.25

VI. CONCLUSIONS

We have presented a new high level synthesis flow for exploiting maximum instruction-level parallelism (ILP) out of applications and mapping it onto a domain-specific custom processor named EPOS. EPOS is a stylized, micro-code driven architecture, supported by ILP-driven architectural features. These enable the exploitation of the statistical parallelism extracted by our synthesis tool. Our front-end compilation in tandem with our scheduling engine generated very efficient schedules for control and data intensive applications. We also evaluated the effectiveness of our forwarding path optimization during binding. We have presented experimental results that show the effectiveness of our synthesis flow in comparison to a previously published work, NISC. EPOS has demonstrated 3.45X performance gain on average compared to NISC. By activating register-file bypassing and predicated operation speculation, we can further improve performance by 25% on average. In our future work we plan to extend our binding engine to optimize RFB and RF access. By combining these optimizations with forwarding minimization we will be able to build extra efficient and low-cost processors.

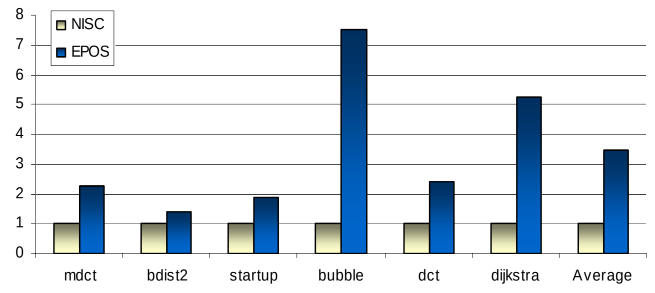


Fig. 6. Execution latency speedup

ACKNOWLEDGMENT

We would like to thank Shane Ryoo, Sain-Zee Ueng and the IMPACT group at the University of Illinois at Urbana-Champaign for their support in leveraging the impact compiler and their invaluable feedback.

This project is partially supported through an NSF grant CCF 07-02501. We used machines donated by Intel.

REFERENCES

[1] C. McNairy and D. Soltis, Itanium 2 Processor Microarchitecture. *IEEE Micro* 23(2): 44-55, 2003.
[2] D. I. August et al., Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture, *ISCA*, 1998.
[3] D. Chen, J. Cong, and J. Xu, Optimal Simultaneous Module and Multi-voltage Assignment for Low Power. *ACM Trans. On Design Automation of Electronic Systems*, Vol. 11, No. 2, 362-386, 2006.
[4] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw Hill, Inc, 1994.
[5] H. Sharangpani and K. Arora, Itanium Processor Microarchitecture. *IEEE Micro* 20(5): 24-43, 2000.
[6] J. W. Sias et al, Field-testing IMPACT EPIC research results in Itanium 2, *ISCA*, 2004.
[7] J. Babb et al, Parallelizing Applications into Silicon, *FCCM*, 1999.
[8] M. Reshadi, et al, Utilizing Horizontal and Vertical Parallelism with a No-Instruction-Set Compiler for Custom Datapaths. *ICCD*, 2005.
[9] M. S. Schlansker and B. R. Rau, EPIC: Explicitly Parallel Instruction Computing. *IEEE Computer* 33(2): 37-45, 2000.
[10] P. Chang et al, Three Architectural Models for Compiler-Controlled Speculative Execution. *IEEE Trans. Computers* 44(4): 481-494, 1995.
[11] S. A. Mahlke et al, Effective compiler support for predicated execution using the hyperblock. *MICRO*, 1992.
[12] S. Gupta, R. Gupta, and N. Dutt, Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Trans. Design Autom. Electr. Syst.* 9(4): 441-470, 2004.
[13] S. Gupta et. al., Using global code motions to improve the quality of results for high-level synthesis. *IEEE Trans. on CAD of Integrated Circuits and Systems* 23(2): 302-312, 2004.
[14] W. W. Hwu et. al., The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *Journal of Supercomputing*, 1993.
[15] http://gelato.uiuc.edu/
[16] K. Yeager, The MIPS R10000 Superscalar Microprocessor, *Micro*, Vol 16, Issue 2, 28-40, 1996.
[17] R. Camposano and W. Wolf. *High-level VLSI synthesis*. Springer-Verlag New York, LLC, 2001.
[18] D. Gajski, N. Dutt, and A. Wu. *High-level synthesis: Introduction to chip and system design*. Kluwer Academic Publishers, 1992.
[19] J. Cong, Y. Fan, and W. Jiang. Platform-based resource binding using a distributed register-file microarchitecture. *ICCAD*, 2006.
[20] K.-H. Lim, Y. Kim, and T. Kim. Interconnect and communication synthesis for distributed register-file microarchitecture. *DAC*, 2007.