A High Performance Application Representation for Reconfigurable Systems

Wenrui Gong Gang Wang Ryan Kastner
Department of Electrical and Computer Engineering
University of California, Santa Barbara
Santa Barbara, CA 93106-9560 USA
{*gong, wanggang, kastner*}*@ece.ucsb.edu*

Abstract

Modern reconfigurable computing systems feature powerful hybrid architectures with multiple microprocessor cores, large reconfigurable logic arrays and distributed memory hierarchies. Mapping applications to these complex systems requires a representation that allows both hardware and software synthesis. Additionally, this representation must enable optimizations that exploit fine and coarse grained parallelism in order to effectively utilize the performance of the underlying reconfigurable architecture. Our work explores a representation based on the program dependence graph (PDG) incorporated with the static single-assignment (SSA) for synthesis to high performance reconfigurable devices. The PDG effectively describes control dependencies, while SSA yields precise data dependencies. When used together, these two representations provide a powerful, synthesizable form that exploits both fine and coarse grained parallelism. Compared to other commonly used representations for reconfigurable systems, the PDG+SSA form creates faster execution time, while using similar area.

1. Introduction

High performance reconfigurable computing systems are extremely complex. These hybrid architectures use more than one billion transistors and consist of multiple microprocessor cores, configurable logic arrays and a distributed memory hierarchy. They allow hardware performance with software flexibility and enable higher productivity [5].

Reconfigurable computing systems are based on standard programmable platforms that allow post-manufacturing customization. The components of these platforms use different types of configuration files. The configurable logic arrays requires a lower level representation; they are programmed at the logic level and largely mimic a hardware design flow. On the other hand, the integrated processor cores require a software design flow. This creates a large amount of freedom for exploring application mappings. At the same time, it introduces enormous complexity to the application designer.

We believe that a common application representation is needed to tame the complexity of mapping an application to state of the art reconfigurable systems. This representation must be able to generate code for any microprocessors in the reconfigurable systems. Additionally, it must easily translate into a bitstream to program the configurable logic array. Furthermore, it must allow a variety of transformations and optimizations in order to fully exploit the performance of the underlying reconfigurable architecture.

In order to achieve high performance, applications mapped to reconfigurable computing systems must generate parallelism during synthesis process. There is a large amount of work dealing with *fine grain parallelism* [16]. Systems supporting fine grain parallelism (e.g. vector and superscalar architectures) have multiple functional units where each unit can perform an independent operation. Fine grain parallelism is employed by issuing an operation to a free functional unit. Techniques exploiting fine grain parallelism are focused mainly on innermost loops.

Coarse grain parallelism is another important technique to improve application performance. Coarse grain parallelism is employed by executing multiple threads (or behaviors) in parallel with occasional synchronization [2]. As a result, coarse grain compiler optimizations focus on parallelization of outer loops. Reconfigurable computing systems feature a novel computing paradigm, which supports both fine and coarse grain parallelism.

A variety of dependence analysis and transformations are used to extract parallelism. In order to gain maximum benefits from these techniques, it is necessary to adopt a good program representation. In our work, we use the program dependence graph (PDG) with the SSA form as a representation for synthesis. The PDG and SSA forms are both common representations in microprocessor compilation. Therefore, the PDG+SSA representation can be

transformed into assembly code, which is used to program the microprocessor core(s) in the reconfigurable system. In this work, we concentrate on synthesizing the PDG+SSA representation to configurable logic array.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 gives a brief introduction to the compilation process. Section 3.1 presents the basic idea of the PDG. Section 3.2 shows how the PDG is extended to a synthesizable program representation. Section 4 describes the synthesis of the PDG+SSA representation to a configurable logic array. Section 5 presents some experimental results. We conclude and give some thoughts on future work in Section 6.

2. Related Work

A number of different program representations have been utilized for the compilation and synthesis of sequential programs to reconfigurable computing systems.

DEFACTO [4] uses the SUIF IR, a *syntax tree* based structure. Several transformations are performed on the syntax tree, including unroll-and-jam, scalar replacement, loop-peeling and array renaming. Most of these transformations are techniques that exploit fine grain parallelism. The DEFACTO project is focused on high-level transforms and directs architectural synthesis using industrial tools.

Mahlke *et al.* [15] proposed using the *hyperblock* to relax the limits of control flow on parallelism and leverage multiple data-paths and functional units in superscalar and VLIW processors. Several projects use similar concepts to exploit fine grain parallelism. The Garp compiler [7] maps standard ANSI C programs to the Garp architecture, which combines a microprocessor core with reconfigurable hardware. The Garp compiler first builds a CFG for each procedure and then creates hyperblocks. These hyperblocks are synthesized to the programmable logic array.

The predicated static single-assignment (PSSA) form, introduced by Carter *et al.* [8], is based on the static single assignment (SSA) form and the notion of hyperblocks. Sea Cucumber [20] - a synthesizing compiler mapping Java byte-code to FPGAs - uses the PSSA to automatically detect fine grain parallelism. CASH [6] is a compiler framework for synthesizing high-level programs into application-specific hardware. It uses the Pegasus representation that augments PSSA using *tokens* to explicitly express synchronization and handle *may-dependence*. Tokens are also used to serialize the execution of consecutive hyperblocks. The projects using the *hyperblock* or the PSSA are mainly focused on finding parallelism in the inner loops, i.e. exploiting fine grain parallelism.

The program dependence graph (PDG), initially proposed by Ferrante *et al.* [12], is a general program representation. The PDG allows traditional optimizations [12], code vectorization [3] and can be used to automatically detect parallelism [9, 13, 18].

A variety of research has been conducted to improve the PDG. These works are mainly focused on incorporating the SSA form and eliminating unnecessary control dependencies. Several program representations using the PDG+SSA have been suggested, such as the program-dependence web (PDW) [17] and the value-dependence graph (VDG) [21].

Our work uses the PDG incorporated with the SSA form. Unlike the PDW, we don't limit the argument number of the ϕ-nodes. This provides more flexibility in synthesis. This representation provides the same ability of exploiting fine grain parallelism as the PSSA or the *hyperblock*. Additionally, it creates coarse grain parallelism since that the transformations on the PDG exploits both loop parallelism and nonloop parallelism.

3. Compilation Process

We focus on the multimedia applications that exhibit a complex mix control and data operations. We assume that the behaviors of these applications are specified using a sequential language. Sequential programming languages are widely used in designs of reconfigurable computing systems. A wide variety of compiler techniques and tools for sequential languages exist and can be leveraged. Furthermore, most programmers are familiar with sequential languages. Therefore, sequential languages are often used to specify tasks and behaviors in embedded system design.

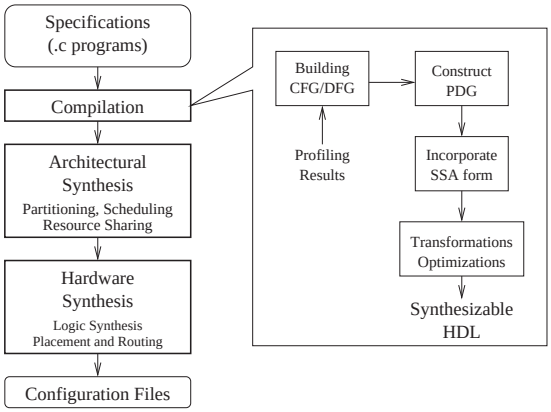

Figure 1. A design flow for reconfigurable systems

Our work focuses on synthesizing sequential programs into reconfigurable computing systems. Figure 1 shows the design flow. Applications are specified in a sequential programming language. The compiler transforms the application into a control/data flow graphs (CDFGs) and integrates profiling information. Then, the CDFG is converted into the PDG+SSA form. After fine and coarse grain parallelism optimizations, the compiler outputs the optimized programs in a register transfer level (RTL) hardware description language. Commercial tools can then be used to synthesize the RTL code into a bitstream to program the

configurable logic array.

3.1. Constructing the PDG

We use the PDG to represent control dependencies. The PDG uses four kinds of nodes - the ENTRY, REGION, PREDICATE, and STATEMENTS nodes. A ENTRY node is the root node of a PDG. A REGION node summarizes a set of control conditions. It is used to group all operations with the same set of control conditions together. The STATEMENTS and PREDICATE nodes contain arbitrary sets of expressions. PREDICATE nodes also contain predicate expressions. Edges in the PDG represent dependencies. An outgoing edge from Node *A* to Node *B* indicates that Node *B* is control dependent on Node *A*.

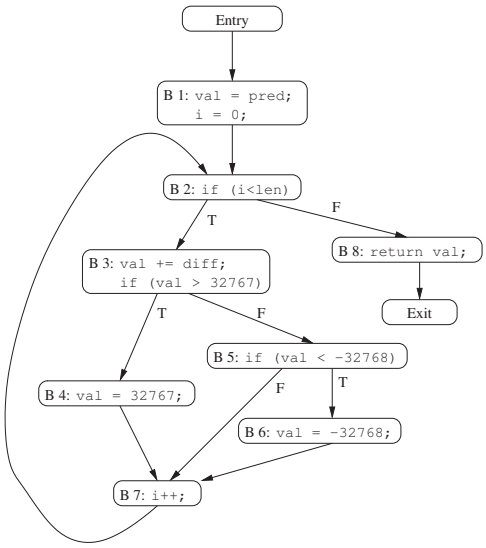

Figure 2. The control flow graph of a portion of the ADPCM encoder application.

The PDG can be constructed from the CFG following Ferrante's algorithm [12]. Each node in the PDG has a corresponding node in the CFG. If a node in the CFG produces a predicated value, there is a PREDICATED node in the PDG; otherwise, there is a STATEMENTS node in the PDG.

A post-dominator tree is constructed to determine the control dependencies. Node *A postdominates* node *B* when every execution path from *B* to *exit* includes node *A* [16]. For example, in Figure 2, every execution path from *B*2 to the *exit* includes *B*8, therefore *B*8 post-dominates *B*2, and there is an edge from node 8 to node 2 in the post-dominator tree (see Figure 3).

Control dependencies are determined in the following manner. If there is an edge from node *S* to node *T* in the CFG, but, *T* doesn't postdominate *S*, then the least common ancestor of *S* and *T* in the post-dominator tree (node *L*) is used. *L* will be either *S* or *S*'s parent. The nodes on the path from *L* to *T* are control-dependent on *S*. For example, there is an edge from 3 to 4 in the CFG and 4 does not postdominate 3. Hence 4 is control-dependent on 3. Using the same intuition, it can be determined that both 7 and 3 are control-dependent on 2.

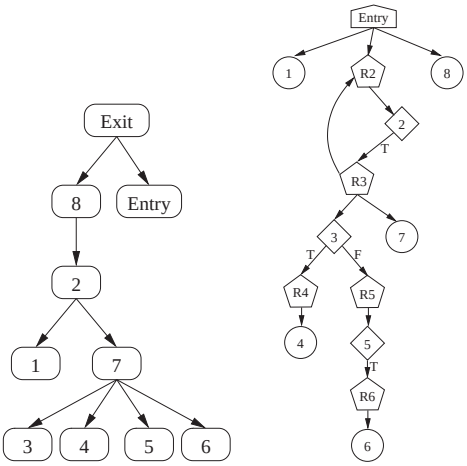

Figure 3. The post-dominator tree and the control dependence subgraph of its PDG for the Figure 2 example.

After determining the control dependencies, REGION nodes are inserted into the PDG to group nodes with the same control conditions together. For example, 3 and 7 are executed under the same control condition $\{2\mathtt{T}\}$. Hence a node *R*3 is inserted to represent $\{2\mathtt{T}\}$, and both 3 and 7 are children of *R*3. This completes the construction of *control dependence subgraph* of the PDG (See Figure 3).

3.2. Incorporating the SSA Form

In order to analyze the program and perform optimizations, it is also necessary to determine data dependencies and model them in the representation. We incorporate the SSA form into the PDG to represent the data dependencies. We model data dependencies using edges between STATEMENTS and PREDICATE nodes.

```
val += diff;
if (val > 32767)
    val = 32767;
else if (val < -32768)
    val = -32768;
```

(a)

```
val_2 = val_1 + diff;
if (val_2 > 32767)
    val_3 = 32767;
else if (val_2 < -32768)
    val_4 = -32768;
val_5 = phi(val_2,val_3,val_4);
```

(b)

Figure 4. Before (a) and after (b) SSA conversion

In the SSA form, each variable has exactly one assignment, and it will be referenced always using the same name. Hence, it effectively separates the values from the location where they are stored. At joint points of a CFG, special ϕ nodes are inserted. Figure 4 shows an example of the SSA form.

The SSA form is enhanced by summarizing predicate conditions at joint points, and labeling the predicated val-

ues for each control edge. This is similar to the PSSA form. In the PSSA form, all operations in a hyperblock are labeled with full-path predicates. This transformation indicates which value should be committed at these join points, enables predicated execution and reduces control height. For example, in Figure 5(a), *val_2* will be committed only if the predicate conditions are {3F,5F}.

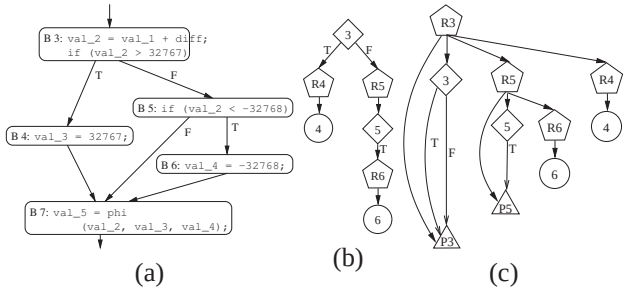

Figure 5. Extending the PDG with the ϕ-nodes

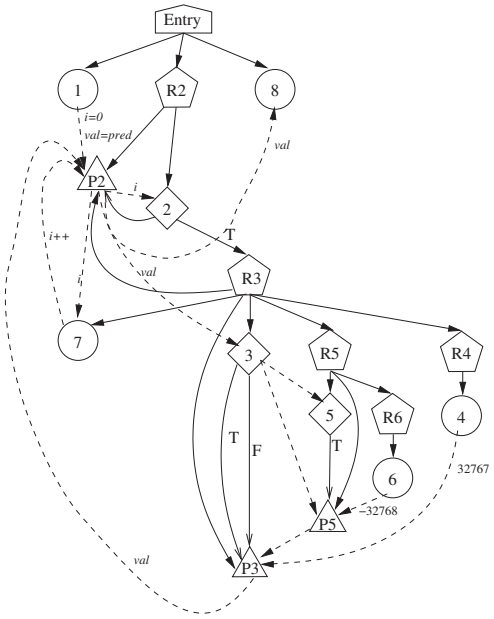

Figure 6. A dependence graph, which is converted to benefit speculative execution, shows both control and data dependence. Dashed edges show data-dependence, and solid ones show control-dependence

In order to incorporate the PDG with the SSA form, a ϕ-node is inserted for each PREDICATE node *P* in the PDG. Figure 5(c) shows that the control dependence subgraph extended by inserting ϕ-nodes. This ϕ-node has the same control conditions as the PREDICATE node, i.e. this ϕ-node will be enabled whenever the PREDICATE node is executed. ϕ-nodes inserted here are not the same as those originally presented in [10]. A ϕ-node contains not only the ϕ-functions to express the possible value, but also the predicated value generated by the PREDICATE node. This determines the definitions that will reach this node. This form is similar to the gated SSA form. However, unlike gated SSA form, this form does not constrain the number of arguments of the ϕ-nodes. Therefore, we can easily combine two or more such ϕ-nodes together during transformations and optimizations.

After inserting ϕ-nodes, data dependencies are expressed explicitly among STATEMENTS and PREDICATE nodes. Figure 6 shows such a graph. Within each node, there is a data-flow graph. Definitions of variables are also connected to ϕ-nodes, if necessary.

3.2.1. Loop-independent and loop-carried ϕ-nodes

There are two kinds of ϕ-nodes, *loop-independent* ϕ-nodes, and *loop-carried* ϕ-nodes. A loop-independent ϕ-node takes two or more input values and a predicate value, and, depending on this predicate, commits one of the inputs. These ϕ-nodes remove the predicates from the critical path in some cases, enable speculative execution, and therefore increase parallelism.

A loop-carried ϕ-node takes the initial value and the loop-carried value, and also a predicate value. It has two outputs, one to the iteration body, and another to the loop-exit. At the first iteration, it directs the initial values to the iteration body if the predicate value is true. At the following iterations, depending on the predicate, it directs the input values to one of the two outputs. For example, in Figure 6, Node *P*2 is a loop-carried ϕ-node. It directs *val* to either *n*8 or *n*3 depending on the predicate value from *n*2. This loop-carried ϕ-node is necessary for implementing loops.

3.2.2. Speculative execution

High performance representations must support speculative execution. Speculative execution performs operations before it is known that they will be needed to execute. In the PDG+SSA representation, this equates to removing control conditions from PREDICATE nodes. Consider the control dependence from Node 3 to R5, i.e. the control path if *val* is less than 32767. This control dependence is substituted by one from Node R3 to R5, which means Node R5 and its successors will be executed before the comparison result in Node 3 becomes available.

4. Transforming to a Synthesizable Hardware Description Language

The PDG+SSA form has natural mapping into a hardware description language (HDL), which can be synthesized using commercial tools to a bitstream to program the configurable logic array.

PREDICATE and STATEMENTS nodes present arbitrary sets of expressions or data-flow graphs (DFGs). In order to synthesize such DFGs into a bitstream, a variety of methods can be utilized. We currently use a one-to-one mapping. It is possible to use a number of different scheduling

and binding algorithms and perform hardware sharing to generate smaller circuits; this is out of the scope of this paper, however we plan on addressing this in future work.

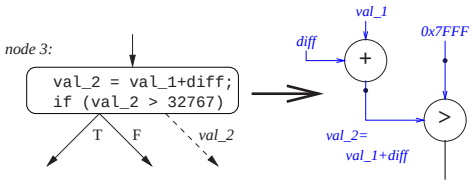

Figure 7. Synthesizing the ϕ-node

Figure 7 shows the synthesis of data-path elements in node 3 of the previous example (see Figure 6). Each operation has an operator and a number of operands. Operands are synthesized directly to wires in the circuit since each variable in the SSA form has only one definition point. Every PREDICATE nodes contains operations that generate predicate values. These predicate values are synthesized to Boolean logic signals to control next-stage transitions and direct multiplexers to commit the correct value.

A loop-independent ϕ-nodes are synthesized to a multiplexer. The multiplexer selects input values depending on the predicate values. For example, as shown in Figure 8, *P*5 is translated to a two-input multiplexer `MUX_P5`, which uses the predicated value from 5 to determine which result should be committed.

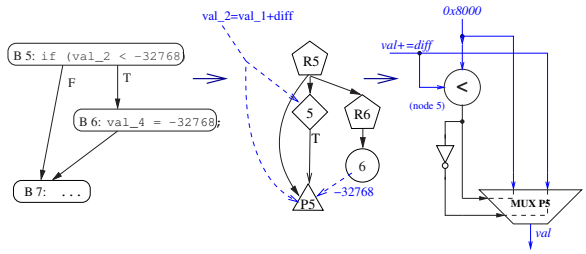

Figure 8. Synthesizing the ϕ-node

A little more work is required to synthesize a loop-carried node since it must select the initial value and the loop-carried value and direct these values to the iteration exit. Using a two-input multiplexer, the initial value and the loop-carried value can be selected depending on the predicate values. A switch is generated to direct the loop-exiting values.

Before synthesizing the PDG to hardware, some optimizations and simplification should be done. For example, unnecessary control dependencies can be removed. Node R4 and R6 in Figure 6 are unnecessary and can be removed. Cascaded ϕ-nodes, such as nodes P3 and P5, can be combined into a bigger ϕ-node with all predicated values. This allows the downstream synthesis tools to choose a proper (possibly cascaded) multiplexor implementation. These ϕ-nodes can also be synthesized directly if necessary i.e. the downstream synthesis tools do not perform multiplexor optimization.

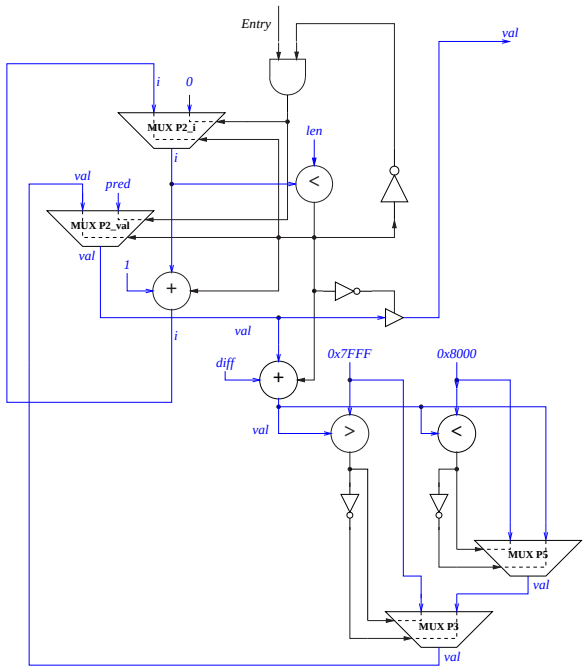

Figure 9. FPGA circuitry synthesized from the above PDG (See Figure 6)

Synthesizing the PDG removes artificial control dependencies. Only those necessary control signals will be transmitted. After synthesis, scheduling should be performed to insert flip-flops to guarantee that correct values will available no matter which execution path is taken.

5. Experimental Results

We use MediaBench [14] as our benchmark suite. More than 100 functions in 6 multimedia applications are tested. Among them, results of 16 functions are reported here. The other non-reported functions exhibited similar behavior. Table 1 shows some statistical information for the reported functions.

The experiments are performed using the SUIF/Machine SUIF infrastructure [1, 19]. SUIF provides a front-end for the C programming language, and Machine SUIF constructs the CFG from the SUIF IR. Using the HALT profiling tool included with Machine SUIF, we import profiling results of the MediaBench applications from the representative input data included with the benchmark suite. We created a PDG pass, which currently performs limited analysis and optimizations.

After constructing the PDG, we estimate the execution time and synthesized area on a configurable logic array. The target architecture is the Xilinx Virtex II Platform FPGA [22]. Based on the specification data of Virtex II FPGA, we get the typical performance characteristics for

	Operations				CFG		PDG		
	#Instr	ALM	Mem	CTI	#N	#Instr/N	R	P	C
func_1	233	148	10	18	31	7.52	32	13	18
func_2	188	128	9	14	24	7.83	25	10	14
func_3	73	52	3	2	5	14.60	5	2	3
func_4	79	51	1	7	13	6.08	15	5	8
func_5	22	15	1	1	3	7.33	3	1	2
func_6	68	51	0	6	10	6.80	10	5	5
func_7	81	55	3	8	13	6.23	13	6	7
func_8	326	250	25	1	3	108.67	3	1	2
func_9	391	306	34	1	3	130.33	3	1	2
func_10	52	36	1	6	10	5.20	12	3	7
func_11	140	104	5	12	18	7.78	19	7	11
func_12	104	72	3	11	17	6.12	18	7	10
func_13	118	85	7	9	14	8.43	17	5	9
func_14	142	104	6	6	11	12.91	11	4	7
func_15	95	54	4	5	9	10.56	11	3	6
func_16	491	336	16	49	67	7.33	77	27	40

Table 1. Statistical information of CFGs and PDGs, including the number of operations, the number of logic and arithmetic operations, memory access, and control transfer instructions; the number of CFG nodes and average instructions per CFG node; the number of REGION nodes, PREDICATE, STATEMENTS nodes in PDGs.

every operation, which is used estimate the performance of the CFG and the PDG.

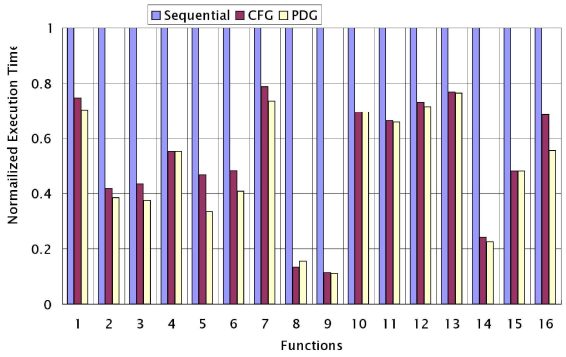

Figure 10. Estimated execution time of PDGs and CFGs

Figure 10 shows the estimated execution time using the PDG representation compared with CFG representation and sequential execution. The PDG and the CFG are 2 to 3 times faster than sequential execution; the PDG is about 7% faster than the CFG. These results use a simple scheduling scheme for estimation. In the CFG, a basic block can be executed when all its predecessors complete their execution. In the PDG, a node is executed once it's control and data dependencies are satisfied.

Figure 11 shows the estimated execution time of PDG+SSA form. Here an aggressive speculative execution is performed. All possible execution paths are taken and the results are committed when the predicated value are available. The results of the PDG+SSA form are on average 8% better compared with the results of the aggressive speculative execution results of the PSSA form.

It is necessary to note that our experimental results do not use all of the optimizations presented in the original

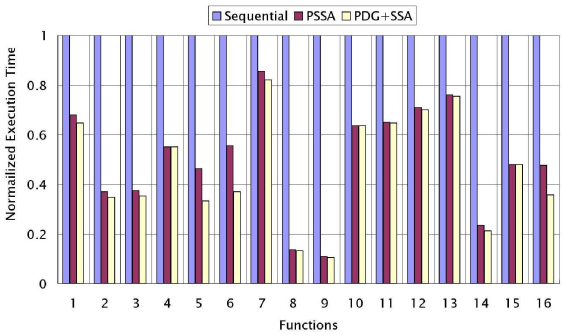

Figure 11. Estimated execution time using aggressive speculative execution

PSSA paper [8]. The projects using the PSSA representation [6, 20] perform other optimizations, including operation simplification, constant folding, dead-code removal and SSA form minimization. Though PDG+SSA form is capable of performing these optimizations, we did not perform these optimizations in our experiments. It is unclear how these optimizations will affect the performance and area results. We intend to look into the use of these optimizations in future work. Our results simply indicate that when using aggressive speculative execution, the PDG+SSA form executes faster than the PSSA form.

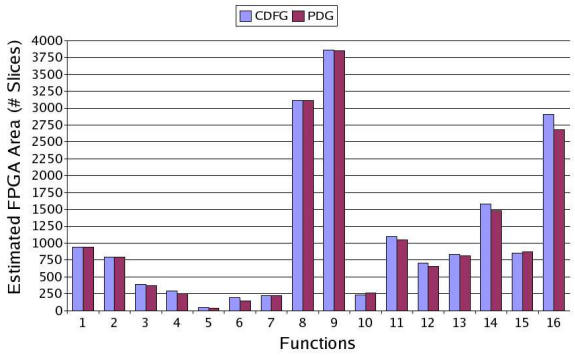

Figure 12. Estimated area of the PDG and CFG representations

Figure 12 shows the estimated number of FPGA slices. The results are estimated based on the one-to-one mapping. Hence, each operation takes a fixed amount of computational resources. The results of the PDG are a little better than those of the CFG, but the difference is small. We did not consider resource sharing. These results are similar to those reported by Edwards [11]. His results show that that the PDG will generate smaller circuits than the CFG for control intensive applications e.g. applications described using the Esterel language.

6. Conclusion

Modern reconfigurable computing systems require synthesis tools to generate configurations to program increas-

ingly complex reconfigurable architectures. The tools require an intermediate representation that allows hardware and software compilation flows. Additionally, it must enable transformations and optimizations that exploit the underlying high performance reconfigurable architecture.

This work showed that an intermediate representation based on PDG+SSA form supports a broad range of transformations and enables both coarse and fine grain parallelism. We described a method to synthesize this representation to a configurable logic array. Experimental results indicate that the PDG+SSA representation gives faster execution time using similar area when compared with CFG and PSSA forms.

In future work, we plan to investigate transformations to create coarse grained parallelism using the PDG+SSA form. Furthermore, we wish to exploit the possibilities of extending our representation to handle other system design languages. For example, it would be interesting to provide interfaces to system level modeling languages, such as SystemC. Also, it would be interesting to augment the PDG+SSA representation with architectural information to provide fast estimation. As part of this estimation, we plan to study the integration of resource sharing and other architectural synthesis techniques into our high level form.

References

[1] G. Aigner, A. Diwan, D. L. Heine, M. S. Lam, D. L. Moore, B. R. Murphy, and C. Sapuntzakis. *An Overview of the SUIF2 Compiler Infrastructure*. Computer Systems Laboratory, Stanford University, 1999.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Karfmann Publishers, San Francisco, CA, 2002.

[3] W. Baxter and H. R. Bauer, III. The Program Dependence Graph and Vectorization. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989.

[4] K. Bondalapati, P. Diniz, P. Duncan, J. Granacki, M. Hall, R. Jain, and H. Ziegler. DEFACTO: A Design Environment for Adaptive Computing Technology. In *Proceedings of the 6th Reconfigurable Architectures Workshop*, 1999.

[5] K. Bondalapati and V. K. Prasanna. Reconfigurable Computing Systems. *Proceedings of the IEEE*, 90(7):1201–17, July 2002.

[6] M. Budiu and S. C. Goldstein. Compiling Application-Specific Hardware. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, 2002.

[7] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp Architecture and C Compiler. *Computer*, 33(4):62–69, April 2000.

[8] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated Static Single Assignment. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 1999.

[9] R. Cytron, J. Ferrante, and V. Sarkar. Experiences Using Control Dependence in PTRAN. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 186–212. The MIT Press, Cambridge, MA, 1990.

[10] R. Cytron, M. Hind, and W. Hsieh. Automatic generation of dag parallelism. In *Proceedings fo the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1989.

[11] S. A. Edwards. High-Level Synthesis from the Synchronous Language Esterel. In *Proceedings of the IEEE/ACM 11th International Workshop on Logic and Synthesis*, 2002.

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–49, July 1987.

[13] R. Gupta and M. L. Soffa. Region Scheduling: An Approach for Detecting and Redistributing Parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–31, April 1990.

[14] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, 1997.

[15] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, 1992.

[16] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[17] K. J. Ottenstein, R. A. Ballance, and A. B. Maccabe. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 1990.

[18] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, 1989.

[19] M. D. Smith and G. Holloway. *An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization*. Division of Engineering and Applied Sciences, Harvard University, July 2002.

[20] J. L. Tripp, P. A. Jackson, and B. L. Hutchings. Sea Cucumber: A Synthesizing Compiler for FPGAs. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, 2002.

[21] D. Weise, R. F. Crew, M. Ernst, and B. Steengaard. Value Dependence Graphs: Representation Without Taxation. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languag*, 1994.

[22] Xilinx, Inc. *Virtex-II Platform FPGAs: Complete Data Sheet*, October 2003.