João M. P. Cardoso, **"Self Loop Pipelining and Reconfigurable Dataflow Arrays,"** in *International Workshop on Systems, Architectures, MOdeling, and Simulation (SAMOS IV)*, Samos, Greece, July 19-21, 2004. *Computer Systems: Architectures, Modeling, and Simulation*, LNCS 3133, **Springer Verlag**, Andy Pimentel and Stamatis Vassiliadis (Eds.), July 2004, pp. 234-243.

Self Loop Pipelining and Reconfigurable Dataflow Arrays

João M. P. Cardoso[1,2]

[1] Faculty of Sciences and Technology, University of Algarve
Campus de Gambelas, 8000 – 117 Faro, Portugal
[2] INESC-ID, Lisbon, Portugal
jmpc@acm.org

Abstract. This paper presents some interesting concepts of static dataflow machines that can be used by reconfigurable computing architectures. We introduce some data-driven reconfigurable arrays and summarize techniques to map imperative software programs to those architectures, some of them being focus of current research work. In particular, we briefly present a novel technique for pipelining loops. Experiments with the technique confirm important improvements over the use of conventional loop pipelining. Hence, the technique proves to be an efficient approach to map loops to coarse-grained reconfigurable architectures employing a static dataflow computational model

1 Introduction

Dataflow machines [1] have been promising to overcome the poor-support of parallelism of von Neumann architectures since the early 70's [2]. However, their envisaged use has been transcended by efforts on augmenting the parallel processing capabilities of traditional processors (*e.g.*, VLIW). There is now a strong believe that it will be very difficult to take full advantage of the Moore's Law using traditional processor architectures. Since dataflow computing is a natural paradigm to process data streams, it is a very promising solution for stream-based computations, which indeed are becoming increasingly important. Some researchers have already focused on synthesizing programs to ASICs behaving in a static dataflow fashion [3]. One of the reasons is the avoidance of centralized control units, which is an ideated goal since the evidence that interconnection delays are becoming preponderating.

Processor arrays, namely wavefront [4] and data-driven arrays [5], have been introduced in the 80's. They devised a scalable and effective fashion to directly support the dataflow computational model and have been revived by some reconfigurable architectures (*e.g.*, KressArray [6]). The dataflow computing model has been used in signal processing and other applications. Recently, research efforts on dataflow computing have been conducted (see, for instance, [7]), especially its usage in reconfigurable computing due to the fact that it naturally supports computing in space. Asyn-

chronous dataflow FPGAs (Field Programmable Gate Arrays) [8] and coarse-grained architectures with dataflow semantics (*e.g.*, WaveScalar [9]) are focus of recent research efforts with encouraging results.

Coarse-grained reconfigurable architectures (see *e.g.*, [10] for information on several architectures) are promising computing platforms. Some of them mix concepts of data-driven arrays with the reconfiguration properties of the programmable logic devices (*e.g.*, FPGAs). Two such examples are the KressArray [6] and the XPP [11]. Although using coarse-grained architectures significant speedups have been achieved, the capability to compile from a high-level imperative programming language, and to still achieving noticeable speedup impact, has not been fully proved, apart from results on mapping specific algorithms. One of the reasons is the reduced focus on researching reconfigurable architectures to unburden the compilation phases and to map more effectively some typical computational structures.

This paper examines some of the most relevant characteristics of the reconfigurable architectures, operating under the dataflow computational model, introduces some compilation techniques to target static dataflow reconfigurable architectures, illustrates some architecture operations to assist compilation, and shows a new loop pipelining technique, named *self loop pipelining* (SLP). Moreover, this paper also aims to address the following questions:

? With respect to other reconfigurable architectures, are data-driven architectures a better target for software compilation?
? What is the impact when using *self loop pipelining*?

This paper is organized as follows. Next section introduces some of the coarse-grained, data-driven, reconfigurable architectures. Section 3 briefly explains some architecture features to support computational structures, and section 4 summarizes compilation techniques for those architectures. Section 5 explains the SLP technique, and focuses on the impact of the technique on a number of benchmarks. Finally, concluding remarks and ongoing and future work issues are sketched in section 6.

2 Data-driven Array Architectures

Data-driven architectures usually use a handshaking protocol to control the data flow, in such a way that the execution of each functional unit (FU) starts when data is present in the required inputs and next result can start to be computed or output (due to previous consumption or absence of output tokens).

Specific hardware implementations can be constructed using FUs or regions of FUs behaving according to the static dataflow model (*e.g.*, [3]). As has been aforementioned, another approach is the use of data-driven reconfigurable arrays (*e.g.*, [11]), either working asynchronous, synchronously, or both. Note, however, that the static dataflow computing model [1] is the simplest to implement in VLSI. There is no strong evidence the complexity required by the dynamic dataflow model [1] is worth to be implemented (it permits to directly map recursive functions, *e.g.*, [12]).

A data-driven array mainly consists of a matrix of N? M PEs and interconnection resources (as an example, see in Figure 1 a simple scheme of the XPP architecture [11]).

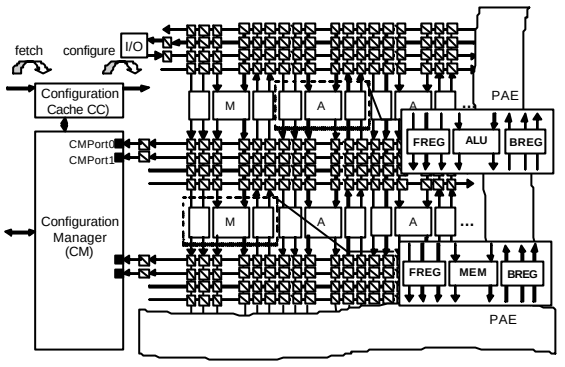

Fig. 1. Simple diagram of the PACT XPP architecture (notice that the picture does not show many architecture's details). PAEs are elements of the architecture that can include an ALU or a memory. The PAEs with memories are located in the left and rightmost columns of the array. Each PAE also includes two more elements: FREG and BREG. These elements can perform special operations and

Dataflow operations, which are implemented by PEs, include usual arithmetic and logic operations, and especial operations to deal with conditional branches.

Conditional branches require BRANCH (SWITCH, DISTRIBUTOR, or DEMUX) and MERGE (JOIN, SELECT, or MUX) operations [1]. BRANCH is used to route data items to one of the two outputs based on a control event (usually named control token). Standard MERGE operations do not have an enabling rule and just output the first data item present in one of the two inputs. There are, however, different implementations of MERGE. One MERGE uses a control signal to select between the two input data tokens and discards the data token (*i.e.*, the token is consumed but not copied to the output) not selected. According to the enabling rule, there are also different MERGE implementations. One only triggers the execution when the control token and the two data tokens are ready, the other one triggers the execution as soon as the control token and the selected data token are ready (this type of evaluation is called lenient in [3]).

Other special operators are specifically used to discard tokens, e.g., the T- and F-Gates used in some dataflow machines, which copy input data to output when the control token has value "true" or when has value "false", respectively [2].

Enhanced dataflow architectures integrate the semantics of imperative programming languages to manipulate array variables (e.g., load/store operations). Two strategies are used for load/store ordering: ordering conducted according to statically labeling of program references (*e.g.*, WaveScalar [9]); ordering explicitly accomplished by control tokens (*e.g.*, XPP [11]). When memories are located in special PEs, array structures are used to access them, and MERGE operations without discard are needed to multiplex data tokens (*e.g.*, XPP). Other architectures use a data sequencer to stream data to the array (*e.g.*, KressArray).

The interconnections are responsible to flow data and control tokens. Their bit-width is a property dependent on the granularity of the PE. Architectures with explicit lines for control events also include 1-bit width interconnections (*e.g.*, XPP). Others

use the data buses to flow either data or control tokens. Note that each interconnection implicitly has lines to implement the handshake mechanism.

Each configuration defines the operations in the PEs and the interconnections among them. Additional units are needed to control array reconfiguration. For efficient support of the configuration flow, architectures may include on-chip configuration manager (CM) and configuration cache (CC), as is the case of the XPP. Such amenities enable efficient and effective implementations of large programs by using temporal partitioning, especially when the number of resources to map a given algorithm exceeds the array resources [13][14].

Table 1 illustrates some of the characteristics of four data-driven arrays: Function Processor [12], KressArray [6], WASMII [16], XPP [11], and WaveScalar [9]. The first three were introduced in the early 90's. They have been pioneer work, as far as reconfigurable computing is concerned. The XPP is a commercial architecture introduced in the late 90's. The WaveScalar is one of the most recent research efforts attempting to build a decentralized dataflow machine and was introduced in the beginning of the 2000's.

Among several distinguishing features, we choose the schemes used to support load/store operations, to map loops, some differences in the operations and interconnect structures of the array and on the static or dynamic dataflow model of computation, as the main representative ones. Some of the arrays use special horizontal and vertical buses (*e.g.*, XPP), others explicitly use PEs for routing and provide interconnections between PEs in a mesh (*e.g.*, KressArray) or in a hexagonal topology (*e.g.*, Function Processor). Although some advantages and disadvantages of using different properties may be enumerated, a study of the impact of those properties, on, *e.g.*, performance, is still required. Notice, however, that with respect to reconfiguration support several differences could be sketched.

Table 1. Data-driven array architectures. (1) on inputs of each cell (configured as FIFOs or LIFOs); (2) Input/output queues, to store data for different waves; (3) A data-sequencer streams data to array; (4) Explicit connections to on- and off-chip memories; (5) Load/store operations can be performed in any PE

Architecture	Dataflow Model	Programming Language	Special buffers in FUs?	Memory semantics
Function Processor [12]	Dynamic	Functional programming language	Yes(1)	No
KressArray [6]	Static	ALE-X (C-based)	No	No(3)
WASMII [16]	Static	Dataflow language (DFC)	No	No
XPP [11]	Static	C or NML (native language)	No	Yes(4)
WaveScalar [9]	Dynamic	C	Yes(2)	Yes(5)

3 Support to Computational Structures

Besides work on using dataflow languages to program data-driven arrays [1], some efforts have been conducted to use imperative programming languages (*e.g.*, [17]). However, lack of specific machine operations to effectively support high-level languages has been one of the major difficulties to attain more efficient compilation results. Although apparently tailored for computations alike the ones described in high-level languages, coarse-grained reconfigurable architectures require further research both on compiler techniques and on operators support. For instance, special functionalities can be directly supported by primitive operations. One of such features permits to implement a counter with a single PE of the architecture (as is the case in the XPP). The counter is one of the operators that truly assists the mapping of high-level languages, specifically well-behaved FOR loops.

Another example is the support of load/store operators in any PE of the architecture (*e.g.*, WaveScalar), without needing to route, for instance, PEs to a port (*e.g.*, with selection capability) of the specific memory cell. One or more buses can be provided to exclusively access memories. Special labeling of the array references in the code [9] and arbitration in order to preserve load/store ordering can be used.

Another issue arises when, in a certain configuration, array resources need to be shared. This needs special control, to route distinct sources by the correct order to the input of the cell being shared, and to route each data item being output from the shared cell to the correct destination. This kind of structures can be implemented by SWITCH and MERGE operators and using control structures to generate the correct event sequences to accomplish the correct paths. The shared resources can be I/O ports for streaming data, which need to be steered to the correct PE (see Figure 2a). For this type of computations, two new operations can be ideated: SE-PAR and PAR-SE. They perform continuously serial to parallel and parallel to serial operations on the input data, respectively. SE-PAR has one input (A) and two outputs (X and Y). The operator repeatedly alternates data on the input to either X and Y (see Figure 2b). PAR-SE has two inputs (A and B) and one output (X). It repeatedly outputs to X the inputs in A and B, in an alternating fashion. Both operators can really assist compilation, fully decentralizes the needed control, without requiring additional resources.

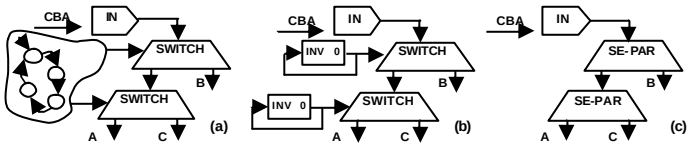

Fig. 2. Routing sequentially each item in a data stream to a distinct destination. Use of SWITCH operators and control structures to create a different path for each item: (a) a centralized; (b) decentralized; (c) naturally decentralized by using SE-PAR operators

MERGE operators with lenient behavior have been already used in [3]. They are important for conditional constructs, because they permit to continue computing as

soon as data produced on the selected branch arrive (no need to wait for the data produced by the other branches).

In this type of architectures pipelining is, as in other computational models, very important. Register stages are added to full balance different paths in order to achieve maximum throughput. When PEs with input/output FIFOs are used, pipeline balancing may in some cases be naturally achieved without adding extra pipeline stages (and thus sophisticated pipeline balancing algorithms are not required). However, a study about the impact of FIFOs' size on the achieved performance for a set of representative benchmarks needs to be carried out. FIFO queues are also important structures for some implementations, especially when a delay of some stages is needed. This can be accomplished using a number of simple FIFOs in sequence, but that may require several PEs. Therefore, their direct support should be considered.

The presented schemes can be directly supported by data-driven arrays to assist the mapping of computational structures described in high-level languages. Moreover, they enclose the necessity to experimentally evaluate some design decisions in order to design a new data-driven array.

As a final remark, when compiling to reconfigurable hardware (see, for instance, [15]), a specific architecture to implement the input algorithm is synthesized. The architecture is usually composed of a data-path and a centralized control unit. Operations performed by the data-path are statically schedule and its execution is then controlled by the FSM (Finite State Machine) generated from the scheduling. Being a centralized control mechanism, it leads to difficulties to achieve the maximum performance, since the complexity to fully pipeline large examples and to tune the timing constraints to be used by place and route tools. Data-driven arrays do not need a centralized control unit, the operations are not statically schedule, and it is the data flow that dynamically imposes the execution of a particular operation (notice, however, that it is possible to statically define an order among operations using control tokens). Both data and control tokens flow concurrently through the array structures, and the implementation therefore naturally exposes fine-grain parallelism and multiple flows of control.

4 Compiling to Dataflow Array Architectures

Mapping computational structures to dataflow architectures is almost direct when straight-line code is input and each operation in the code can be directly implemented by a PE of the target architecture. The handshaking mechanism permits to abstract the mapping from the timing details associated when the computational structures are implemented using a data-path and a centralized control unit (timing-driven model).

When mapping conditional constructs (such as *if-then-else* statements) MERGE and SWITCH operations can be used. These two operations can also be used to implement loops. SWITCH operations are used to select the data flow through the loop structures (during iterations) or to path it to the structures beyond the loop (after loop completion).

To map imperative programming languages to a dataflow machine, the input computational structures can be transformed to the Program Dependence Web (PDW) [18], a representation that extends the Static Single Assignment (SSA) form [19] and the Program Dependence Graph (PDG) [20]. The PDW contains all the needed information for control-, data-, and demand-driven interpretation, and thus it can be used to generate the DFG akin to the required dataflow structure.

Selection points are explicitly represented in the SSA-form by ? -functions. Those points can be directly implemented with MERGE operations with discard. The PDW uses the Gated Single Assignment (GSA) to generate the control conditions. Instead of using only the SSA ? -functions, the PDW uses three types of functions (μ, ?, and ?). μ-functions are used to represent selection points between loop carried values and loop initializations (MERGE operation). ?-functions are used to control forward data flow (MERGE operation). Finally, ? -functions are used to control passage of values out of loop bodies (*i.e.*, they are used to forward final data values after loop completion). ? -functions can be translated to SWITCH nodes. Operations to forward a copy or to discard the input data item may also be used (*e.g.*, T- and F-gate). Note that as opposite to non dataflow modes, where operations using a certain assignment are scheduled to time steps where data are already available, here we have to ensure that only data that must be used arrive to destination.

To enable the firing of some operations, control tokens are used, either directly (*i.e.*, as a form of predicate execution controlled by guards) or as control mechanisms to cease the data flow. Architectures with PEs with firing rules enabled by special control inputs can almost directly implement predicated execution (*e.g.*, XPP). When these type of firing rules are not directly supported, special operators can be added to enable/disable the data flow to destinations. Nevertheless, when speculative execution is used enable/disable firing is not needed as long as the data generated in paths not taken are discarded.

In the end, some transformations on the DFG may be necessary to be ready for placement and routing on the data-driven array. Pipeline balancing is usually performed during the place and route phase (*e.g.*, XPP).

Note however that for efficient compilation several optimizations are still required (see, for instance, [3]), such are the cases of software pipelining and elimination of redundant memory accesses (*e.g.*, inter-iteration register promotion [21]). A novel dataflow specific optimization, called *loop decoupling*, has been introduced in [3]. It slices a loop into multiple independent loops that may run ahead from each other. To ensure that no data-dependences are violated they use a token generator operator. It dynamically controls the dependence distance between decoupled loop iterations.

5 Self Loop Pipelining (SLP)

One of the most efficient design optimizations is pipelining. Pipelining is a form of overlapping different steps of computations. The use of pipelining leads usually to

significant performance improvements. With respect to loops, *software pipelining* is a fundamental technique to improve throughput.

As far as dataflow computing is concerned, efficient loop execution has been achieved through *dataflow software pipelining* [22]. The approach uses balancing techniques to exploit maximum throughput. Balancing is achieved by the use of a certain number of register stages or FIFOs in each arc of the dataflow model.

Consider the example in Figure 3a. Figure 3b shows a pipelined implementation. The CNT module represents a counter which starts at a given number, increments by a certain quantity until a certain limit is not exceeded. Using a data-driven model with handshaking, the counter only furnishes a new value if the previous one has already been consumed.

To enable optimal software pipelining, full balancing of paths is required (see Figure 3b), *i.e.*, the counter indexing consecutive elements of the arrays A, B and C, requires that the two paths arriving to the destination memory where array C is located are balanced. The two paths are related to the operations computing the data items to be stored in the array C, and to the address generation structure. For balancing, elements behaving as simple registers have been inserted.

Now we briefly introduce *self loop pipelining* (SLP), a novel technique for pipelining loops. Figure 3c shows the main concept. The original centralized counter, responsible for the control iterations of the FOR loop, is duplicated and now two decentralized counters are responsible for the loop control behavior. The counters are decoupled and synchronize indirectly due to the data flow. As is depicted, there are now two independent paths furnishing the index value (i) to access array elements. Note that another correct SLP implementation would use three counters (one for each memory).

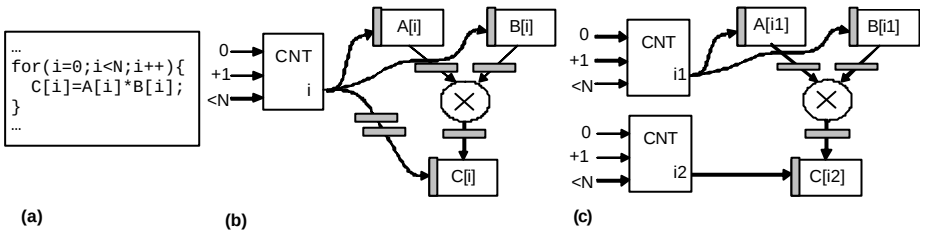

Fig. 3. Loop pipelining on data-driven machines: (a) simple example - each array is mapped to a distinct memory; (b) traditional loop pipelining; (c) the proposed *self loop pipelining* technique. Rectangles in gray represent pipeline stages

With SLP, loops are naturally executed in a pipelining fashion. The technique is based on duplicating the cyclic hardware structures responsible to loop control (in the example is implemented using a counter, but usually can be implemented using a hardware cyclic structure), in order they are autonomously executed, with synchronization being naturally achieved by the data flow. It can be applied to all kind of loops (DO-WHILE, WHILE, and FOR), and also to nested loop structures. Using SLP, innermost loops with conditional constructs can also be pipelined without conservative

loop pipelining implementations (which are usually based on the maximum critical path length of the loop body). The technique requires fewer resources for balancing (*i.e.*, fewer registers or smaller FIFOs) and less sophisticated balancing efforts than previous software pipelining techniques.

We have semi-automatically mapped some benchmarks to the XPP [11] using SLP. The architecture performs each PE operation and communicates data between elements (*i.e.*, PEs or interconnection registers) in a single clock cycle.

The results are now compared with loop pipelining implementations automatically achieved with the XPP-VC compiler [17]. When applying SLP to various DSP kernels (*max, auto correlation, weighted vector sum, block move, gourad, and median*) [23], performance improvements are achieved. With SLP we obtain from 1.2% to 68.4% fewer execution cycles, and even fewer needed PEs for most examples.

The improvements achieved with SLP have origins in the more relaxed *pipeline balancing* requirements and in the unneeded matching of branches on conditional constructs to achieve the maximum throughput.

Our ongoing work focuses on more experiments and on extending our compiler [15] to target static dataflow machines, including the use of the SLP technique.

6 Conclusions

This paper discusses some interesting reconfigurable array architectures computing in a dataflow fashion. Promising research efforts to compile imperative software languages to those kinds of architectures are also introduced. One of the techniques is a novel form of loop pipelining, named *self loop pipelining*.

Self loop pipelining can be used to pipeline loops in data-driven architectures based on the ready-acknowledge principle of operation. It involves replication of the hardware structures responsible for the control of loop iterations. The proposed technique has been applied for mapping a number of benchmarks to the XPP. Results, achieving performance improvements and fewer required resources, strongly confirm its importance. Ongoing and future work aims compilation techniques to automatically apply the technique. Future work should also embrace experiments with static dataflow models with input/output FIFOs in the functional units.

Acknowledgments

This work is in part supported by the Portuguese Foundation for Science and Technology (FCT) - FEDER and POSI programs - under the CHIADO project. The author gratefully acknowledges the donation by PACT XPP Technologies, Inc, of the XPP development suite (XDS) software.

References

1. A. H. Veen, "Dataflow machine architecture," in *ACM Computing Surveys*, Vol. 18, Issue 4, Dec. 1986, pp. 365-396.
2. J. B. Dennis, D.P. Misunas, "A computer architecture for highly parallel signal processing," in *Proc. ACM National Conference*, ACM, New York, Nov. 1974, pp. 402-409.
3. M. Budiu, *Spatial Computation*, Ph.D. Thesis, CMU CS Technical Report CMU-CS-03-217, Dec. 2003.
4. S. Y. Kung, et al., "Wavefront Array Processors - Concept to Implementation," in *IEEE Computer*, vol. 20, no. 7, July 1987, pp. 18-33.
5. I. Koren, et al., "A Data-Driven VLSI Array for Arbitrary Algorithms," in *IEEE Computer*, October 1988, pp. 30-43.
6. R. Hartenstein, R. Kress, and H. Reinig, "A Dynamically Reconfigurable Wavefront Array Architecture," in *Int'l Conference on Application Specific Array Processors (ASAP'94)*, Aug. 22-24, 1994, pp. 404-414.
7. W. A. Najjar, E. A. Lee, and G. R. Gao, "Advances in the dataflow computational model," in *Parallel Computing*, vol. 25, 1999, Elsevier Science B.V., pp. 1907-1929.
8. J. Teifel, and R. Manohar, "Highly Pipelined Asynchronous FPGAs," in *ACM Int'l Symposium on Field-Programmable Gate Arrays (FPGA'04)*, Monterey, CA, USA, Feb. 2004.
9. S. Swanson, et al., "WaveScalar," In *36th Annual Int'l Symposium on Microarchitecture (MICRO-36)*, Dec., 2003.
10. R. Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective," In *Int'l Conf. on Design, Automation and Test in Europe (DATE'01)*, Munich, Germany, March 12-15, 2001, pp. 642-649.
11. PACT XPP Technologies, Inc., "The XPP White Paper," Release 2.1.1, March 2002, http://www.pactxpp.com.
12. J. Vasell, J. Vasell. "The Function Processor: A Data-Driven Processor Array for Irregular Computations," in *Future Generation in Computer Systems*, 8(4), 1992, pp. 321-335.
13. João M. P. Cardoso, "Loop Dissevering: A Technique for Temporally Partitioning Loops in Dynamically Reconfigurable Computing Platforms," in *10th Reconfigurable Architectures Workshop (RAW'03), Nice, France, April 2003*, IEEE Computer Society Press.
14. João M. P. Cardoso, and Markus Weinhardt, "From C Programs to the Configure-Execute Model," in *Proc. of the Design, Automation and Test in Europe Conference (DATE'03)*, Munich, Germany, March 3-7, 2003, IEEE Computer Society Press, pp. 576-581.
15. João M. P. Cardoso, and Horácio C. Neto, "Compilation for FPGA-Based Reconfigurable Hardware," in *IEEE Design & Test of Computers Magazine*, March/April, 2003, vol. 20, no. 2, pp. 65-75.
16. X. Ling, et.al., "WASMII: An MPLD with Data-Driven Control on a Virtual Hardware," in *Journal of Supercomputing.*, Vol.9, No.3, 1995, pp.253-276.
17. João M. P. Cardoso, and Markus Weinhardt, "XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture," in *12th Int'l Conference on Field Programmable Logic and Applications (FPL'02)*, LNCS 2438, Springer-Verlag, 2002, pp. 864-874.
18. 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 *ACM Conference on Programming Language Design and Implementation (PLDI'90)*, 1990, pp. 257–271.
19. R. Cytron, et al., "Efficiently Computing static single assignment form and the control dependence graph," In *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, October 1991, pp. 451-490.
20. J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," in *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, July 1987, pp. 319-349.
21. S. Carr, D. Callahan, and K. Kennedy, "Improving register allocation for subscripted variables," In *Proc. of ACM Conference on Programming Language Design and Implementation (PLDI'90)*, June 1990, ACM Press.
22. G. R. Gao, *A Code Mapping Scheme for dataflow Software Pipelining*, Kluwer Academic Publishers, 1991.
23. Texas Instruments, Inc., "TMS320C6000™ Highest Performance DSP Platform," 1995-2003, http://www.ti.com/sc/docs/products/dsp/c6000/benchmarks/62x.htm#search