Trade-offs in Mapping High-level Dataflow Graphs onto ASIPs

Vladimír Guzma*, Shuvra S. Bhattacharyya†, Pertti Kellomäki*, and Jarmo Takala*
*Department of Computer Systems
Tampere University of Technology, Tampere, FI-33720, Finland
{vladimir.guzma, pertti.kellomaki, jarmo.takala}@tut.fi
†Department of Electrical and Computer Engineering
University of Maryland, College Park, MD 20742, USA
ssb@umd.edu

***Abstract*— Data-flow based design environments bring advantages of specification, validation and synthesis to embedded systems design by decoupling computation from transfer of data. The former is performed by actors, and data transfer between actors and an execution order of actors is determined by scheduling and buffering strategies. In this work, we examine code sizes and cycle counts resulting from combinations of scheduling and buffering techniques. The experiments were carried out by designing an application specific instruction-set processor streamlined for each of the benchmarks, using a codesign environment called TCE. We also show what additional overhead is introduced when an architecture implemented using our approach is employed for an application outside its targeted domain.**

I. Introduction

The dataflow programming model represents a program as a set of tasks (actors), and data dependencies (FIFO queues) between actors. Individual actors consume data from their inputs and produce data on their outputs when they are executed. The functionality of whole program is defined by the functionality of the individual actors together with the semantics of their interconnections. In area of digital signal processing (DSP), the applications often work on a streams of data. Therefore, the scheduled dataflow graph needs to be executed in an iterative manner, running within a loop (often an infinite loop) without deadlocks, and using only a finite amount of physical memory.

The synchronous dataflow (SDF) model [1] supports these requirements well for an important class of signal processing applications. With the application written as an SDF graph, the actual work is performed by the actors, while a *schedule* for the graph defines the order in which the actors actors are executed, and also defines requirements for buffer management between actors. Schedules and their associated buffer management requirements in general add some overhead to the execution time, code size and consequently instruction memory, and data memory requirements of an SDF-based application.

In this work, we model benchmarks as SDF graphs using the dataflow interchange format [2] (DIF), which is a tool for developing and experimenting with DSP-oriented dataflow models of computation. We use the DIF-to-C tool [3] to synthesize C code for each of the benchmarks using different combinations of SDF-based scheduling and buffering strategies.

Our generation of ASIPs is done using the TTA Codesign Environment [4] (TCE). We use the TCE compiler to compile synthesized C code for different benchmarks onto different ASIP instances, and we use the TCE simulator [5] to obtain the count of executed instructions.

We also explore the relative quality of critical and non-critical applications in this framework. Specifically, we select one benchmark as being *critical* (highest priority for optimized implementation), and tune the processor architecture to minimize execute cycle count for the critical application. We then recompile other (*non-critical*) benchmarks for the derived architecture, and measure the overhead observed when executing the non-critical benchmarks on a processor that was not tuned specifically for those applications.

Our experiments demonstrate important trade-offs and interactions among SDF-based applications; SDF techniques for scheduling and buffer management; and critical and non-critical application support in ASIPs. Our work also demonstrates a novel design flow that integrates SDF techniques from the DIF environment with ASIP technique from the TCE environment.

II. Related Work

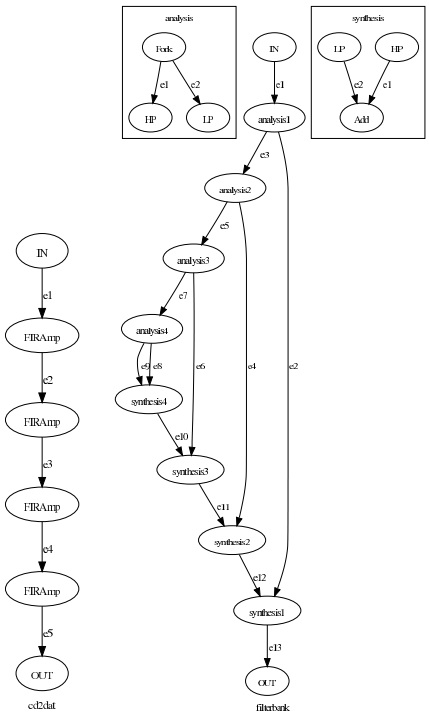
Other approach to introduce dataflow programming to DSP area is presented in SPEX language extension [6]. SPEX adds constructs to programming language (C++ in presented work) to allow programmers to describe inherent parallelism within a DSP system, including describing streaming computation and communication patterns of DSP systems.

Focusing on distributed control and memory, Kahn process networks are used in [7], as a method for programming high-throughput multimedia on a platforms consisting of multiple microprocessors and reconfigurable components. From application written in subset of Matlab, Kahn process network is automatically derived.

In [8] authors present a high-level heterogeneous functional specification and verification part of system for modeling and simulation of embedded systems, *El Greco*. Their system allows specifications in forms of cyclo-static dataflow (CSDF)

1-4244-2542-6/08/$20.00 ©2008 IEEE

Authorized licensed use limited to: University of Maryland College Park. Downloaded on March 13, 2009 at 13:53 from IEEE Xplore. Restrictions apply.



(a) cd2dat (b) filterbank

Fig. 1. Graphical representation of SDF benchmarks

and hierarchical finite state machines, with ability to nest models at any level.

III. Experimental Setup

In our experiments, we use four benchmarks from the DSP domain: multi-stage CD-DAT (cd2dat) and DAT-CD (dat2cd) sample rate conversion, a four-level tree-structured filter bank performing the bi-orthogonal wavelet decomposition (fb4bwd), and a JPEG encoder subsystem with RGB-YCbCr, 2d-DCT, Quantization and ZigZag sequencing (jpeg). Filter bank and CD-DAT are shown in Fig. 1 as a SDF applications. We used the DIF-to-C software synthesis framework [3] to produce C code for the applications for all compatible combinations of buffering and scheduling strategies. Our choice of buffering strategies includes:

- buffer sharing [9] (abbreviated BS)
- modified circular buffering [10] (abbreviated C)
- static read-write pointer resetting [1] (abbreviated SRW)
- in-place buffer merging for in-place actors [11] in JPEG (abbreviated IPBM)

Our choice of scheduling strategies includes:

- dynamic programming post optimization SDPPO [9] (abbreviated SD)
- acyclic pairwise grouping of adjacent nodes [12] and DPPO post optimization (abbreviated AD)
- flat scheduling, using topological sorting (abbreviated F)
- recursive procedure call based multiple appearance schedule [13] (abbreviated RPC)

The filter bank benchmark is modeled using a hierarchical SDF graph (with the actors for analysis and synthesis defined as SDF subgraphs), so we apply scheduling on the hierarchical as well as flattened graphs. This gives us two alternative schedules — one that respects the hierarchy of the original application specification, with each SDF subgraph scheduled independently, and another that is not necessarily constrained to follow this hierarchy. We use the TTA Codesign Environment (TCE) [4] to generate target architectures and compile and simulate each benchmark. TCE allows the designer to select the number of function units and what operations each unit performs; the number of register files, including the number of their read and write ports and the number of registers in each of them; the number of transport buses; and how function units and register files are connected to each bus. We use TCE with disabled inlining of procedures to get a clear view of how many cycles are spent in synthesized functions in the code generated by DIF-to-C.

We start by finding an architecture for each benchmark group, determining the minimal cycle count required for each of the benchmarks to process its sample input, and compiling each of the benchmarks for such an architecture. Since the architecture does not change when combinations of scheduling and buffering strategies change for a particular benchmark group, the intra-actor code scheduling is performed in the same way and the number of clock cycles spent inside each actor is unchanged. Therefore in section IV, we only show cycle counts spent in synthesized functions of each benchmark, and leave out the cycle counts for the actors themselves. The synthesized functions are where the actual differences due to choices of scheduling and buffering strategies shows. We also present the code size for the whole application (total) and the code size for the synthesized functions only (synthesized) for each of the benchmarks and strategies combination. We also estimate the area required for each architecture [14]. This estimate does not include the I/O function unit, since I/O is implementation specific. Then we perform experiments related to application criticality, as described in Section IV-B. Specifically, we take the JPEG benchmark (image processing) to be critical, and the cd2dat, dat2cd, fb4bwd benchmarks (audio processing) to be non-critical, and evaluate the overhead — due to lack of dedicated support — incurred when executing the non-critical benchmarks.

IV. Experimental Results

In this section, we first show counts of executed instructions as well as static code sizes for the applications and their synthesized *main* functions. The synthesized *main* functions embody the parts of code that are generated automatically by DIF-to-C; the remaining code is taken from the actor library components associated with the applications. Afterwards, we show results for cd2dat, dat2cd and the filter bank application compiled for an architecture that has been optimized for jpeg, and we show the increases in cycle count and code size resulting from executing these applications with such a non-critical status.

Authorized licensed use limited to: University of Maryland College Park. Downloaded on March 13, 2009 at 13:53 from IEEE Xplore. Restrictions apply.

TABLE I

CYCLE COUNTS AND NUMBERS OF INSTRUCTIONS

(a) cd2dat

	cycles	synthesized	total
BS-AD, BS-SD	18684	892 (23%)	3933
BS-RPC	16884	5698 (65%)	8739
C-AD	20918	742 (19%)	3772
C-F	34757	738 (19%)	3777
C-RPC	25412	2315 (43%)	5350
SRW-AD, SRW-SD	16912	2130 (41%)	5166
SRW-RPC	22535	1897 (38%)	4938

(b) dat2cd

	cycles	synthesized	total
BS-AD, BS-SD	9836	726 (19%)	3763
BS-RPC	10105	1489 (33%)	4530
C-AD	15134	497 (14%)	3534
C-F	20900	590 (16%)	3628
C-RPC	17445	1421(31%)	4458
SRW-AD, SRW-SD	9816	726 (19%)	3757
SRW-RPC	10700	1456 (32%)	4489

(c) fb4bwd

Hierarchical	cycles	synthesized	total
BS-AD, BS-RPC	5712	2079 (40%)	5127
BS-SD	6042	2258 (42%)	5316
C-AD, C-RPC	6851	2248 (42%)	5288
C-F	8330	3286 (60%)	5388
SRW-AD, SRW-RPC	6075	2077 (40%)	5117
SRW-SD	6087	2273 (43%)	5325
Flat	**cycles**	**synthesized**	**total**
BS-AD, BS-RPC	4208	2351 (43%)	5403
BS-SD	4164	2346 (43%)	5392
C-AD, C-RPC	5957	1978 (39%)	5023
C-F	9475	2803 (48%)	5851
SRW-AD, SRW-RPC	4160	2314 (43%)	5369
SRW-SD	4216	2390 (44%)	5441

(d) jpeg

	cycles	synthesized	total
BS-AD, BS-RPC	8124	603 (27%)	2214
BS-SD	8124	605 (27%)	2216
C-AD, C-RPC	10508	734 (31%)	2341
C-F	19783	1387 (46%)	2996
SRW-AD, SRW-RPC	7926	591 (27%)	2192
SRW-SD	7926	600 (27%)	2201
IPBM-AD	7714	565 (26%)	2148

TABLE II

CYCLE COUNTS AND NUMBERS OF INSTRUCTIONS FOR JPEG ARCHITECTURE TARGET

(a) cd2dat

	cycles	synthesized	total
BS-AD, BS-SD	19615	952 (22%)	4327
BS-RPC	17547	5943 (64%)	9315
C-AD	22028	794 (19%)	4168
C-F	36756	803 (19%)	4180
C-RPC	27765	2469 (42%)	5848
SRW-AD, SRW-SD	17677	2223 (39%)	5610
SRW-RPC	23427	2008 (37%)	5388

(b) dat2cd

	cycles	synthesized	total
BS-AD, BS-SD	10452	777 (19%)	4156
BS-RPC	10773	1585 (32%)	4959
C-AD	16916	539 (13%)	3922
C-F	22637	643 (16%)	4027
C-RPC	19623	1529 (31%)	4901
SRW-AD, SRW-SD	10429	779 (18%)	4167
SRW-RPC	10898	1568 (32%)	4947

(c) fb4bwd

Hierarchical	cycles	synthesized	total
BS-AD, BS-RPC	6334	2202 (39%)	5586
BS-SD	6311	2395 (41%)	5789
C-AD, C-RPC	7225	2366 (41%)	5758
C-F	8863	3355 (57%)	5876
SRW-AD, SRW-RPC	6311	2181 (39%)	5567
SRW-SD	6339	2412 (42%)	5809
Flat	**cycles**	**synthesized**	**total**
BS-AD, BS-RPC	4377	2454 (42%)	5835
BS-SD	4345	2457 (42%)	5849
C-AD, C-RPC	6488	2144 (38%)	5538
C-F	9475	2966 (46%)	6354
SRW-AD, SRW-RPC	4336	2420 (41%)	5805
SRW-SD	4398	2501 (43%)	5884

A. Optimized architecture for each benchmark group

Table I(a) and Table I(b) show cycle counts and code sizes for the cd2dat and dat2cd benchmarks. Table I(c) shows the same kind of data for fb4bwd, but with two sets of data corresponding to hierarchical and flattened schedules (see Section III). Table I(d) shows results for jpeg.

In some cases, the structure of the SDF graph causes different scheduling strategies to produce same schedule. In particular, for cd2dat and dat2cd, dynamic programming post optimization (SD) and pairwise grouping of adjacent nodes with DPPO post optimization (AD) produce the same schedule. It can be seen from the results that the synthesized code contributes significantly to the code size of whole application. For the jpeg and filter bank cases, the AD scheduling and recursive procedure call based multiple appearance scheduling produce the same schedule. For the cd2dat, dat2cd and jpeg benchmarks written as flat SDF graphs, this contribution is between 14% to 65%. Buffer sharing with recursive procedure call (BS-RPC) caused the largest increase in code size for the chain-structured SDF graphs of the cd2dat and dat2cd benchmarks. The more complex topology of the jpeg benchmark caused the largest overhead for the circular buffering with flat scheduling (C-F) strategy. For all three of the benchmarks, C-F added the largest overhead to the cycle count.

From the results, we see that the smallest overhead in cycle count does not correspond to the smallest overhead in code size. Buffer sharing with dynamic programming post optimization (BS-SD) appears to be a strategic choice in regards to this trade-off. For fb4bwd-hierarchical (the hierarchical version of fb4bwd), the synthesized code contributes between 40% and 60% of the total code size. When the flattening strategy is applied, the hierarchy of SDF graphs is flattened before scheduling starts, and therefore, only one SDF graph is scheduled. The relative contribution of synthesized code is generally smaller, in this case — between 39 and 48% for fb4bwd. In both cases, the C-F strategy combination causes the largest code size increase, due to the complex SDF topology that is associated with the application. This strategy also adds the largest overhead to the executed cycle count.

Authorized licensed use limited to: University of Maryland College Park. Downloaded on March 13, 2009 at 13:53 from IEEE Xplore. Restrictions apply.

Most effective, in terms of minimal increase in code size and cycle count, seem to be the BS-SD and SRW strategy combinations. Furthermore, the results show that the circular buffering strategy, with any of the scheduling techniques, consistently causes larger increase in cycle count than buffer sharing or static read-write pointer resetting.

The architecture derived for jpeg has an area estimate of 48 kgates, and the architectures for cd2dat, dat2cd and fb4bwd all have area estimates of about 90 kgates. Significant similarity results in the architectures for cd2dat, dat2cd and fb2bwd. This is caused by the use of floating point computation in the benchmarks, and consequent use of software floating point emulation.

B. Recompiled benchmarks for the critical 'jpeg' architecture

Table II(a) shows cycle counts and code sizes for the cd2dat benchmark compiled for an architecture that has been designed specifically for jpeg. Table II(b) and Table II(c) show analogous results for dat2cd and fb4bwd, respectively. Effects of different combinations of scheduling and buffering techniques on code size and cycle count are similar to the results from Tables I(a), I(b), and I(c). However, recompiling benchmarks as non-critical applications causes up to 12% increases in code size and cycle count compared to the corresponding *native* architectures for these applications. Figures 2(a), 2(b), and 2(c) show graphical comparisons of cycle count and synthesized code size overhead for cd2dat, dat2cd, and fb4bwd, respectively.

V. Conclusion

In this paper, we have demonstrated relationships and trade-offs involving SDF scheduling techniques, SDF buffer management techniques, ASIP implementation, and application criticality. Our work has integrated the SDF techniques of the DIF design tool with the ASIP techniques of the TCE environment, and we have carried out extensive experiments on practical DSP applications using this integrated approach. Our results indicate various relevant trends — for example, that use of circular buffering with flat scheduling adds the highest overhead in terms of cycle count, and for complex topologies, this combination also appears to add relatively high code size overhead. More broadly, we have demonstrated a new methodology for exploring trade-offs involving various key design considerations when mapping SDF graphs onto ASIP platforms. Interesting directions for future work include developing more integrated tool support and automation for this methodology, and applying it to other kinds of high level dataflow transformations and architectural platforms.

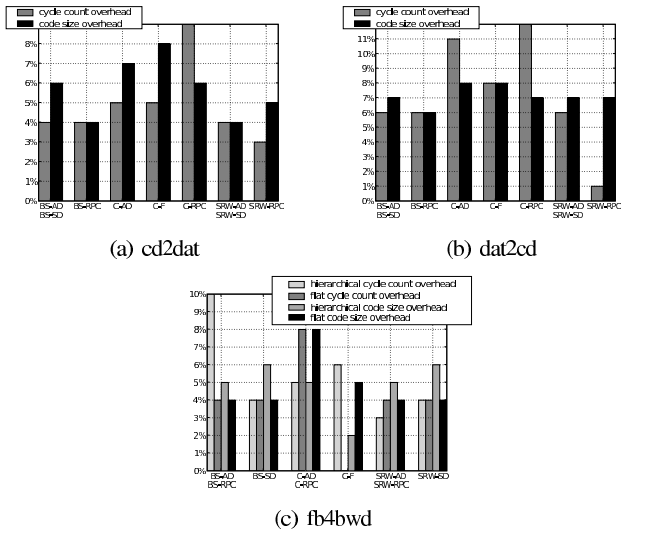


Fig. 2. Cycle count and code size increase when compiled for 'jpeg' target architecture

References

[1] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987.

[2] C. Hsu, F. Keceli, M. Ko, S. Shahparnia, and S. S. Bhattacharyya, "DIF: An interchange format for dataflow-based design tools," in *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, Samos, Greece, July 2004, pp. 423–432.

[3] C.-J. Hsu, M.-Y. Ko, and S. S. Bhattacharyya, "Software synthesis from the dataflow interchange format," in *SCOPES '05: Proceedings of the 2005 workshop on Software and compilers for embedded systems*. New York, NY, USA: ACM, 2005, pp. 37–49.

[4] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala, "Codesign toolset for application-specific instruction-set processors," in *Proc. Multimedia on Mobile Devices 2007*, 2007, pp. 65 070X–1 – 65 070X–11, http://tce.cs.tut.fi/.

[5] P. Jääskeläinen, "Instruction Set Simulator for Transport Triggered Architectures," Master's thesis, Department of Information Technology, Tampere University of Technology, Tampere, Finland, P.O.Box 553, FIN-33101 Tampere, Finland, Sep 2005, \tthttp://tce.cs.tut.fi/.

[6] Y. Lin, Y. Choi, S. Mahlke, and T. Mudge, "A parametrized dataflow language extension for embedded streaming systems," in *Int. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS VIII)*, July 2008, pp. 10–17.

[7] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System design using Kahn process networks: The Compaan/Laura approach," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2004, p. 10340.

[8] J. Buck and R. Vaidyanathan, "Heterogeneous modeling and simulation of embedded systems in El Greco," in *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign*. New York, NY, USA: ACM, 2000, pp. 142–146.

[9] P. K. Murthy and S. S. Bhattacharyya, "Shared buffer implementations of signal processing systems using lifetime analysis techniques," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 2, pp. 177–198, February 2001.

[10] S. S. Bhattacharyya and E. A. Lee, "Memory management for dataflow programming of multirate signal processing algorithms," *IEEE Transactions on Signal Processing*, vol. 42, no. 5, pp. 1190–1201, May 1994.

[11] S. S. Bhattacharyya and P. K. Murthy, "The CBP parameter — a module characterization approach for DSP software optimization," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 38, no. 2, pp. 131–146, September 2004.

[12] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[13] M. Ko, P. K. Murthy, and S. S. Bhattacharyya, "Compact procedural implementation in DSP software synthesis through recursive graph decomposition," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, Amsterdam, The Netherlands, September 2004, pp. 47–61.

[14] T. Pitkänen, T. Rantanen, A. G. M. Cilio, and J. Takala, "Hardware cost estimation for application-specific processor design," in *SAMOS*, ser. Lecture Notes in Computer Science, T. D. Hämäläinen, A. D. Pimentel, J. Takala, and S. Vassiliadis, Eds., vol. 3553. Springer, 2005, pp. 212–221.

Authorized licensed use limited to: University of Maryland College Park. Downloaded on March 13, 2009 at 13:53 from IEEE Xplore. Restrictions apply.