An Integrated ASIP Design Flow for Digital Signal Processing Applications

(Invited Paper)

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*—Application specific instruction set processors (ASIP) allow designers to optimize the architecture of an embedded processor to meet the specific demands of a particular application. A complementary form of customization is provided by domain-specific models of computation (MoCs), which can expose the high level structure of applications that is useful for various kinds of optimizing design transformations. One such MoC is Synchronous Dataflow (SDF), which is used increasingly in the design and implementation of signal processing applications. In this paper, we develop an integration of SDF- and ASIP-oriented design flows, and use this integrated design flow to explore trade-offs in the space of hardware/software implementations. We also explore an approach to ASIP implementation in terms of "critical" and "non-critical" applications, which allows designers to tune the degree of specialization for a targeted ASIP. Our results show that single ASIP processor tuned for pair of critical applications saves 26% to 50% of area required for implementations of two applications on separate ASIPs and non-critical applications runs on such processor with in worst case 4.5% overhead for our selection of benchmarks.**

I. Introduction

Implementing complex DSP algorithms in imperative programming languages (such as C) is prone to human errors and requires thorough testing before such applications can be deployed. Requirements for real time performance, deadlock avoidance and finite physical memory make this problem even harder. Therefore, domain-specific models of computation that capture high level structure of applications are often used to verify correctness of designs, determine memory requirements and automate synthesis of applications. In addition, various kinds of design transformation can be performed on high level computation models, which are not possible at lower levels.

One such model that captures a variety of DSP applications properties is Synchronous Dataflow (SDF) [1]. As with other dataflow programming models, an application is represented in SDF as a set of actors (tasks) and data dependencies between actors (FIFO queues) with the amount of data consumed and produced defined for each actor. Individual actors can be trivial operations, as well as complex computational blocks. In addition, hierarchical SDF allows actors to be SDF graphs themselves, thus allowing for incremental design from single actors to complex application. The behavior of a complete program is therefore defined by the functionality of individual actors and their respective interconnections.

In the SDF model, the order in which actors are executed is defined by the *schedule* of actors and data dependencies between actors are defined by *buffer management*. After appropriate scheduling and buffering strategies are selected, the application can be synthesized automatically and evaluated. In our previous work, we analyzed the execution time overhead and the code size overhead of the synthesized code [2]

In comparison, while programming in an imperative language, a programmer can write applications using a library of functions (equivalent to actors), while manually taking care of data dependencies between functions and the order in which functions are called. This code represents addition to code size and execution times spent in called functions. However, in case the resulting application does not fit into the given requirements, the whole process needs to be repeated.

In embedded DSP designs, there are often situations when several critical applications need to run in sequence, processing input with real time requirements for a given number of times, followed by non-critical applications with no real-time requirements (such as collecting and processing data from sensors within a given time interval, followed by broadcasting data to receivers [3]). Using a stock DSP processor for such purpose could result in a costly design, with the selected DSP processor too powerful for the given set of tasks, eventually wasting energy, or in an insufficient design with some applications missing their real time requirements. The use of ASIP processors is therefore a viable solution to minimize energy consumption and meeting requirements.

In addition, the use of programmable ASIPs allows for extension of the base instruction set with customized instructions, supported with specific hardware resources [4], [5]. Automation of search for such instruction set extensions brings to design the best of both worlds: flexibility of programmable processor, tuned to meet critical requirements of power and timing, together with ASIC design of accelerators, for profitable combinations of instructions. Optimizing an ASIP processor for the whole set of such applications could lead to an unnecessarily complex and energy inefficient design,

when optimized also for non-critical applications. Designing several processors for each application or group of applications leads to inefficiency too, increasing the overall cost of such system, in terms of area as well as energy.

In this paper, we argue that it is viable alternative to optimize a processor for a given set of critical applications, which then meet their real time requirements, and reuse the processor for non-critical applications. These applications perform sub-optimally in terms of cycle count, but sufficiently for the purpose. We show that such a solution, while possibly leading to a larger processor than any individual application would require, still results in smaller area and energy requirements than would a set of processors for each individual application, or a subsets of applications.

In this work, we use the Dataflow Interchange Format [6] to describe applications as SDF graphs, DIF-to-C tool [7] to synthesize application code, and the TTA-based Codesign Environment [8] to generate ASIPs from the synthesized code, and evaluate application performance and processor cost.

The rest of this paper is organized as follows: in Section II we discuss previous work on domain specific models of computation and DSP and ASIP, in Section III we present our solution to this problem, in Section IV we describe the experimental setup, in Section V we present the results and finally in Section VI we conclude the work and outline further research ideas.

II. Related work

In this section, we discuss approaches that start from high level domain specific models of computation and lead to implementations, as well as several aspects of ASIP design with relation to ASIC and instruction set extensions.

A. From domain-specific models of computation to implementation

The SPEX language extension [9] adds language constructs in C++ to allow programmers to describe inherent parallelism within a DSP system. The language features describe streaming computation and communication patterns of DSP systems. In contrast to fully static dataflow approaches, the parametrized dataflow computation model of SPEX allows limited control flow changes in computation, by describing dataflow with a set of parameters. Once the description is complete, the compiler iterates through all the combinations of parameters, producing individual SDF graphs and scheduling them using existing dataflow scheduling algorithms. A prescheduled SDF is selected and executed in run-time based on the actual parameters. The drawback of this approach is that the number of SDFs generated can grow with addition of parameters and therefore the amount of instruction memory required grows.

In [10] the authors present a method for programming high-throughput multimedia on a platform consisting of multiple microprocessors and reconfigurable components using Kahn process networks, exploiting distributed memory and control properties. For an input application, written in subset of Matlab, the correct by construction Kahn process network can be automatically derived and mapped onto a target platform consisting of CPU and FPGA (Compaan/Laura approach).

In [11] the 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) and hierarchical finite state machines, with the ability to nest models at any level.

In [12] the authors present a system for automatic design space exploration, performance evaluation and system generation using SystemC based library, SysteMoC. This synthesizable subset of SystemC allows describing and simulating communicating actors. The models of computation supported by SysteMoC ranges from homogeneous synchronous dataflow [1] to Kahn process networks [13].

In [14] the authors present a programming language and compilation infrastructure designed for streaming systems, *StreamIt*. Effective mapping of StreamIt applications to uniprocessors, multicore architectures and workstation clusters is also provided.

B. Automated application-specific instruction set extension for ASIP

In [15] the authors present an automated compilation flow for detection and generation of application-specific instructions for ASIPs, based on pattern detection, followed by instruction set selection guided by a cost function taking into account occurrence, speedup and area costs. As a last step, the data flow graph is mapped into selected patterns to minimize total latency using binate covering.

In [16] the authors present an algorithm for automatic selection of application-specific instructions with hardware constraints. The instruction selection problem is formulated as a global ILP problem, with minimization of execution time as an objective function.

III. Data flow and ASIP

The Dataflow Interchange Format (DIF) [6] is a textual language designed to model the semantics of graphical design tools, based on dataflow graphs, for DSP systems. In addition to the language itself, the DIF package also contains implementations of algorithms that operate on models described in the DIF language. In particular, DIF has built-in support for synchronous dataflow (SDF) models, cyclo-static dataflow (CSDF) model, heterogeneous and single-rate dataflow, restricted versions of SDF. Support for Boolean dataflow (BDF) and parametrized dataflow (PSDF) is also provided.

Hierarchical models are allowed in DIF, in particular dataflow graph nodes (also called actors) are often dataflow graphs themselves.

Figure 1 shows a DIF specification of a hierarchical SDF graph of a filter bank application and Fig. 2 shows the corresponding SDF graph.

The DIF-to-C package [7] provides a software synthesis framework for automatic generation of C code implementations from specifications programmed in the DIF language.

```
sdf analysis1 {
  topology {
    nodes = Fork, HP, LP;
    edges = e1(Fork, HP), e2(Fork, LP); }
  interface {
    inputs = input : Fork;
    outputs = output1 : HP, output2 : LP; }
  attribute datatype { e1 = "float"; e2 = "float"; ... }
  production { e1 = 1; e2 = 1; output1 = 1; output2 = 1;}
  consumption { e1 = 2; e2 = 2; input = 1; }
  actor Fork {
    computation = "Fork";
    input = input;
    output1 = e1;
    output2 = e2;
    length = 1; }
    ...
}
sdf analysis2 {  basedon {analysis1;} }
sdf analysis3 {  basedon {analysis1;} }
sdf analysis4 {  basedon {analysis1;} }
sdf synthesis1 {...}
sdf synthesis2 { basedon {synthesis1;} }
sdf synthesis3 { basedon {synthesis1;} }
sdf synthesis4 { basedon {synthesis1;} }

sdf filterbank {
  topology {
    nodes = WR, A1, A2, A3, A4, S1, S2, S3, S4, WW;
    edges = e1 (WR,A1), e2(A1,S1), ... , e13(S1, WW); }
  production { e1 = 1; }
  consumption { e13 = 1; }
  attribute datatype { e1 = "float"; e2 = "float"; ... }
  refinement {analysis1 = A1; input : e1; output1 : e2;
    output2 : e3; }
  refinement {analysis2 = A2; input : e3; output1 : e4;
    output2 : e5; }
    ...
  refinement {synthesis1 = S1; input1 : e2; input2 : e12;
    output : e13;}
  refinement {synthesis2 = S2; input1 : e4; input2 : e11;
    output : e12;}
    ...
}
```

Fig. 1. Hierarchical SDF specification of filterbank application written in DIF language

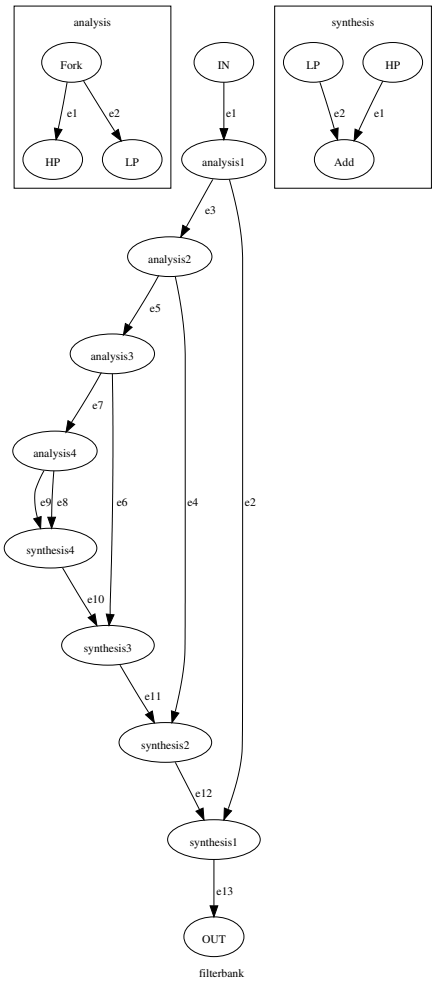

Fig. 2. Hierarchical SDF graph of filterbank application (analysis and synthesis subgraphs are instantiated 4 times)

During the synthesis phase, scheduling of actor execution, as well as buffer management takes place. The DIF package provides a variety of strategies for both, scheduling and buffering.

We have investigated the overhead of various combinations of scheduling and buffering algorithms for several of the DIF specifications in our previous work [2]. Our results show that the overhead of synthesized buffering and scheduling algorithms in terms of code size varies with different combinations of algorithms, ranging from 19% to 65%. The largest overhead was typically achieved, for nontrivial SDF graphs, with combination of circular buffering and flat scheduling (C-F). The smallest overhead was typically found for buffer-sharing and dynamic programming post-optimization SDPPO (BS-SD), combination of strategies used in this paper, and combinations using static read-write pointer resetting (SRW). Similarly, the largest overhead on execution cycle count was typical for C-F combination and the smallest overhead with BS-SD and SRW combinations.

Once software synthesis from DIF specification to C code is complete, our method follows with designing an application specific instruction set processor for the synthesized DIF specification.

We use the TTA codesign framework (TCE) [8] for semi-automated or fully automated design of ASIP processors. In particular, we use design space exploration, seeded with synthesized C code and a minimal architecture. During the exploration process, the explorer repeatedly attempts to extend the processor with various resources, or removes unused or rarely used resources and evaluates the speed of application on such a processor as well as estimates the energy needed [17]. After the exploration process completes, the designer is able to select the architecture based on restrictions given on speed, area or energy. While in this work we concentrate on creation of pairs of applications that works in sequence, the TCE framework also supports compiler assisted multithreading [18].

IV. Experimental setup

In our experiment, we first attempted to find an *optimized* ASIP design for each of the benchmarks (the selected benchmarks are presented in Table I(a)). The characteristics of benchmarks are such that *cd2dat*, *dat2cd* and *fb4bwd* use floating point operations extensively, *jpeg* only uses integer arith-

TABLE I
BENCHMARK APPLICATIONS SPECIFIED IN HE DIF LANGUAGE

(a) Single applications

cd2dat	multi-stage CD-DAT
dat2cd	multi-stage DAT-CD
fb4bwd	four-level filter bank performing
jpeg	encoder subsystem
sar	synthetic aperture radar

(b) Pairs of applications as hierarchical SDF graphs

cd2dat+dat2cd	multi-stage CD-DAT + multistage DAT-CD
dat2cd+fb4bwd	multi-stage DAT-CD + four-level filter bank
fb4bwd+jpeg	filter bank + jpeg encoder subsystem
jpeg+sar	jpeg encoder subsystem + synthetic aperture radar

metic, and *sar* uses string processing as well as floating point arithmetic. Once each of the ASIPs was found, we estimated the area required for its implementation. Our implementation uses an emulation library for floating point operations. In the second step, we attempted to generate ASIPs for combinations of benchmark applications, we call them critical (combinations are listed in Table I(b)). We take advantage of DIF support for hierarchical SDF graphs and create a new SDF graph that combines two existing applications in sequence. We then evaluate the overhead in terms of area such an ASIP has, compared to single implementations of ASIPs for each of the benchmarks selected as critical, and to the sum of critical ASIPs areas. We also attempt to schedule non-critical applications for such an ASIP and explore the overhead in cycle counts that execution on such a non-optimized architecture causes. For each of the benchmarks and combination of benchmarks we selected dynamic programming post-optimization SDPPO as the scheduling strategy and buffer sharing as the buffering strategy. In our previous experiment we concluded that this combination provides good results in terms of code size overhead and cycle count overhead [2] of synthesized code.

V. RESULTS

We used the design space explorer to find the best performing architecture (possibly largest) for each of the synthesized benchmarks and pairs of benchmarks. In Table II(a) we show the resulting execution cycle counts and area estimates for each of the benchmarks from Table I(a). In Table II(b) we show the resulting cycle counts (second column) and area estimates (third column) for each of the benchmark pairs from Table I(b). In the fourth column in this table we list sums of the cycle counts for each of the benchmarks when executed as a single application on its best performing architecture (as shown in Table II(a)), and column five lists sum of the areas for respective benchmark implementations. It is notable that cycle counts of pairs of benchmarks are very close to the sum of cycle counts of individual benchmarks, within range of -5% to +2%, actual improvement for 3 out of 4 combinations. In addition, the area required for pairs with

TABLE II
CYCLE COUNTS AND AREA (GATES) FOR BENCHMARKED APPLICATIONS

(a) Single benchmarks with best architecture

Benchmark	Cycles	Area
cd2dat	12677847	98471
dat2cd	6481104	98989
fb4bwd	394232	98332
jpeg	401815	83613
sar	1114888095	104474

(b) Pairs of benchmarks with best architecture compared to the sum of cycles and areas for best *native* architecture

Benchmark	Cycles	Area	$\sum$ cycles	$\sum$ areas
cd2dat+dat2cd	20204326	98537	19158951	197460
dat2cd+fb4bwd	6707529	99565	6875336	197321
fb4bwd+jpeg	800772	134662	796047	181945
jpeg+sar	1115943570	127757	1115289910	188087

similar characteristics, such as extensive use of floating point in case of *cd2dat+dat2cd* and *dat2cd+fb4bwd* requires area similar to the area required for each of the single participants. Therefore, a combination of such applications requires only 50% of the area that is otherwise required, if each application is implemented individually.

When the benchmarks paired together have very different characteristics, such as *fb4bwd+jpeg* and *jpeg+sar*, the area savings for an implementation with a cycle count close to the sum of the cycle counts of participating benchmarks are a bit smaller when compared to the previous case. In particular, *fb4bwd+jpeg* saves 26% of area required for implementation of each benchmark as individual application, and *jpeg+sar* combination saves 33% of the sum of the areas of the participants. Once the best performing architectures for all the selected pairs of critical applications have been found, we schedule the remaining non critical applications for the architectures. In Table III(a) we show the cycle counts for applications considered non critical for *cd2dat+dat2cd* and the respective ratio compared to cycle counts on their *native* architectures. Table III(b) shows the same results for non critical applications in case of *dat2cd+fb4bwd*, Table III(c) results for non-critical applications in case of *fb4bwd+jpeg* and finally, Table III(d) shows the same results for non critical applications in case of *jpeg+sar*.

It is notable that for architectures optimized for pairs of critical benchmarks, non critical ones can be executed with very small overhead, from less than 0% to 4.5% in the worst case, compare to 0% to 12% when only single application is selected as critical [2].

VI. CONCLUSIONS

In this paper we have presented an implementation methodology that leads from an application specification in high level model of computation, Synchronous Data-flow Graph, to an implementation as an application specific instruction

TABLE III
CYCLE COUNTS FOR NON CRITICAL BENCHMARKS EXECUTED ON BEST ARCHITECTURE FOR CRITICAL PAIR

(a) Best architecture for *cd2dat+dat2cd*

Benchmark	Cycles with cd2dat+dat2cd	% of native cycles
fb4bwd	412131	104.5%
jpeg	408836	101.7%
sar	1143438281	102.5%

(b) Best architecture for *dat2cd+fb4bwd*

Benchmark	Cycles with dat2cd+fb4bwd	% of native cycles
cd2dat	12813103	101%
jpeg	408812	101.7%
sar	1133582942	101.7%

(c) Best architecture for *fb4bwd+jpeg*

Benchmark	Cycles with fb4bwd+jpeg	% of native cycles
cd2dat	12696323	100.1%
dat2cd	6497134	100.2%
sar	1120715667	100.5%

(d) Best architecture for *jpeg+sar*

Benchmark	Cycles with jpeg+sar	% of native cycles
cd2dat	12683352	100%
dat2cd	6482591	100%
fb4bwd	393823	100.1%

set processor. By combining two tools, the DIF specification language with DIF-to-C synthesis framework and the TCE environment for creation of ASIPs, we can map data-flow designs into ASIP processors in straightforward way.

In addition, we have shown that in case of applications run in sequence, it is feasible to design single ASIP processor for combinations of such applications instead of designing single ASIP processor for each of them. Such a configuration saves significant amounts of area (from 50% for applications with similar characteristics to 26% for applications with different characteristics) without adding significant penalty on cycle counts for critical applications. Non critical applications with similar characteristics to critical ones, such as use of floating point arithmetic, can also be run on such a processor with small overhead on execution cycle counts (4.5% cycle count overhead is the worst case for our benchmarks).

Interesting directions for future research includes application of this technology for other kinds of dataflow graphs and dataflow graph transformations, exploring possibilities of compiler-assisted multithreading during scheduling of dataflow graphs, as well as more integrated tool support for automation of this methodology.

ACKNOWLEDGMENT

This work was supported by the Academy of Finland, project 205743, and the Finnish Funding Agency for Technology and Innovation under research funding decision 40163/07.

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] V. Guzma, S. S. Bhattacharyya, P. Kellomäki, and J. Takala, "Trade-offs in mapping high-level dataflow graphs onto ASIPs," in *Int. Symp. System-on-Chip*, Nov 2008.

[3] C. Strydis, C. Kachris, and G. N. Gaydadjiev, "ImpBench: A novel benchmark suite for biomedical, microelectronic implants," in *Int. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS VIII)*, July 2008, pp. 82–91.

[4] M. K. Jain, M. Balakrishnan, and A. Kumar, "ASIP design methodologies: Survey and issues," in *In Proceedings of the IEEE / ACM International Conference on VLSI Design. (VLSI 2001*, 2001, pp. 76–81.

[5] K. Keutzer, S. Malik, and R. Newton, "From ASIC to ASIP: The next design discontinuity," in *ICCD*, January 2002. [Online]. Available: http://www.gigascale.org/pubs/204.html

[6] 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.

[7] 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.

[8] 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/.

[9] 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.

[10] 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.

[11] 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.

[12] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, and J. Teich, "A SystemC-based design methodology for digital signal processing systems," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 15–15, 2007.

[13] G. Kahn, "The semantics of simple language for parallel programming," in *Proceedings of IFIP Congress*, 1974, pp. 471–475.

[14] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *CC '02: Proceedings of the 11th International Conference on Compiler Construction*. London, UK: Springer-Verlag, 2002, pp. 179–196.

[15] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 2004, pp. 183–189.

[16] C. Galuzzi, E. M. Panainte, Y. Yankova, K. Bertels, and S. Vassiliadis, "Automatic selection of application-specific instruction-set extensions," in *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2006, pp. 160–165.

[17] 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.

[18] P. Jääskeläinen, P. Kellomäki, J. Takala, H. Kultala, and M. Lepistö, "Reducing context switch overhead with compiler-assisted threading," Proceedings of The 3rd International Workshop on Embedded Software Optimization (ESO 2008), to appear, 2008.