Automatic Verification of Source Code Transformations on Array-Intensive Programs: Demonstration with Real-life Examples

K.C. Shashidhar *Maurice Bruynooghe*
Francky Catthoor *Gerda Janssens*

Report CW 401, May 2008

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Automatic Verification of Source Code Transformations on Array-Intensive Programs: Demonstration with Real-life Examples

K.C. Shashidhar   *Maurice Bruynooghe*
Francky Catthoor   *Gerda Janssens*

Report CW 401, May 2008

Department of Computer Science, K.U.Leuven

Abstract

We have recently developed a method for verification of source code transformations applied on array-intensive programs typically found in signal processing and high performance computing applications. The method is based on checking the equivalence of the original and the transformed programs belonging to a decidable class that is still of practical interest. In this report, using a prototype implementation, we demonstrate the applicability of the method in practice by verifying transformations on fragments of source codes from several real-life applications. Since most of such application codes do not, at first, lie within the class of programs that our method allows, we make use of some code pre-processing methods, the most important among them being the dynamic single-assignment conversion method, and discuss implications of their use on the transformation verification problem.

1 Introduction

Source code transformations are increasingly becoming necessary while programming for high performance and low power computing and communicating systems. The transformations provide the programmer an opportunity to take advantage of the *algorithm-level* knowledge of the program and apply global optimizations that are usually not in the scope of present day optimizing compilers. In the domain of embedded-processor systems the application of code transformations is particularly common and has been widely advocated as indeed essential (cf. [3, 4, 5, 9, 19]).

This trend, though positive as far as developing better optimized programs is considered, has on the flip side increased the burden on verification of correctness of the transformed program. This is because, the transformations, though complex and subtle, are usually applied manually or, at best, using ad-hoc transformation tools and, needless to say, such application is often prone to errors. Programmers have responded to this by increasing stress on simulation-based testing of the transformed programs. Since testing is time-consuming, inconclusive and lacking in automatic debugging support, there is a need for automatic verification of the transformed program with respect to the original program.

We have developed a method [13, 14] to address this need in the context of optimization of code kernels of DSP applications targeted to low power embedded-processor based systems where transformations are mainly applied in order to optimize references to data memory. The method is based on checking the equivalence of the original and the transformed programs and where the decidability is ensured by restricting the programs to a class, that is defined by the common code characteristics found in the application domain of our interest. The class of programs that our method can handle have the following properties:

1. *Dynamic single-assignment:* Every memory location is written only once. Optimizing compilers use the *static single-assignment* (SSA) form [6] to facilitate optimizations which still can write the same array element several times. This is not the case with *dynamic single-assignment* (DSA) form; it eliminates all false dependencies. Methods for conversion to DSA are described in [10, 17]. We also require that functions are free from side-effects.

2. *Piecewise-affine expressions:* Subscripts in the arrays and expressions in the bounds of the `for`-loops are all piece-wise affine in the iterator variables of the enclosing `for`-loops. Additionally, the expressions can also include operators like `mod`, `div`, `max`, `min`, `floor` and `ceil`. This allows representing the addressing relationships between elements of arrays as affine inequalities in integers and makes it possible to use well-understood dependence tests (for example, the Omega test [12]) to solve those systems.

3. *Static control-flow:* There are no data-dependent `while`-loops in the programs. We assume that data-dependent `while`-loops have been converted to `for`-loops with worst-case bounds and a global if-condition on its body; and the data-dependent if-conditions in the program have been converted into data dependencies by using if-conversion [2].

4. *No pointer references:* Programs are free from pointer references. Pointer-to-array conversion methods (for example, [15]) can be used here.

The class is not unduly restrictive for the application domain. In fact, it is advantageous to bring programs into such a form before applying global trans-

formations as this form creates more freedom for the transformations and the tools used for guiding the transformations can do a better job [5].

For original and transformed programs lying in the allowed class, our method is able to prove equivalence of programs that are related through any combination of transformations from the categories below:

- Loop transformations: both, structure preserving and modifying,
- Expression propagations: compiler optimizations like invariant code motion, copy/constant propagation, common sub-expression elimination/ introduction (when it is less expensive to recompute than to fetch), etc.; and
- Algebraic transformations: those relying on the algebraic properties of fixed-point data-types (ignoring overflow) like, associativity, commutativity, etc.

1.1 Equivalence Checking Method in Brief

The equivalence checking method that our tool implements is presented in [13, 14]. Here we provide only a brief overview. The tool essentially checks the sufficient condition for the equivalence of the two ADDGs. Based on pairs of corresponding data dependence paths in the two ADDGs, it has two parts, viz., the two paths have (1) identical computation; and (2) identical relationship between the elements of the output and the input arrays that are at the either end of the paths. The tool identifies corresponding paths on the fly by way of a *synchronized* traversal of the two ADDGs. During the traversal, the operators in the computation provide points of synchronization and relationship from the elements of the output array to the point is kept updated. When the traversal successfully reaches an input array on a path, the computation on the two paths is guaranteed to be identical (modulo algebraic transformations) and all that remains is to check the relationship between the elements of the output and the input arrays for that path. If this holds, the traversal proceeds to check the remaining paths, until all the possible corresponding paths are exhausted. Due to algebraic transformations, it is possible that a legal transformation still has a mismatch at an operator. Then appropriate operations are invoked in order to reduce the ADDGs to a normal form at that operator. Also, an ADDG can have cycles, they correspond to recurrences. When a cycle is detected, an operation is invoked that avoids stepping through the cycle.

2 Tool Flow

Our prototype program equivalence checking tool implements the scheme shown in Fig. 1. The programs are first subject to a pre-processing stage which chains tools as shown in Fig. 2. This involves: (1) conversion to *dynamic* single assignment form, i.e., removal of all false dependencies; (2) *if-conversion*, i.e., removal of any data-dependent control-flow; and (3) in-lining of functions in order to handle inter-procedural transformations.

We take the original and the transformed program pairs from real-life application design context and discuss the scheme for checking their equivalence using our method.

The equivalence checking scheme is as shown in Figure 1. The input to the checker are the texts of an original program and a program obtained by applying one or more of loop and data-flow transformations on it, called the transformed program. The checker also allows an *optional* set of inputs. These help either

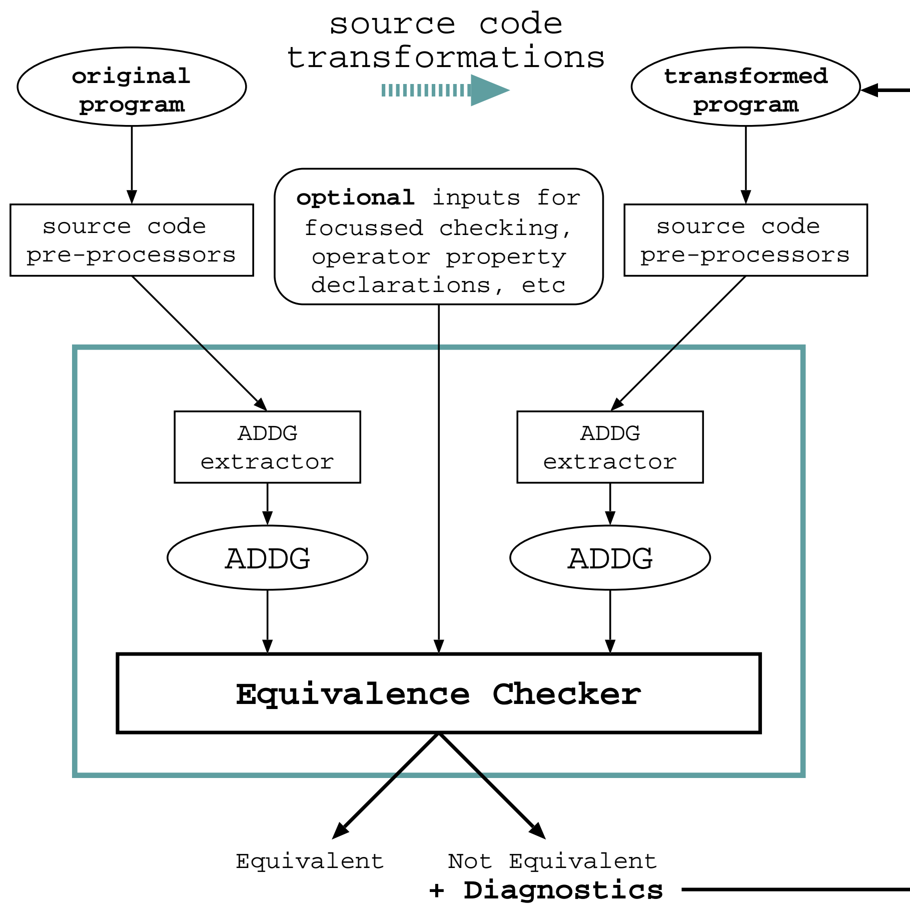

Figure 1: The verification and debugging scheme.

reduce the work for the checker by providing the focus of interest or supply it with additional information to handle algebraic transformations. The following specific options are allowed by our current implementation:

1. output variables, and optionally, their definition domains,
2. declaration of some of the intermediate variables as input variables,
3. whether statement-level or operator-level transformations have been applied, and
4. declaration of arithmetic properties of the operators in the input programs (only if algebraic transformations have been applied).

The first two options help focus the equivalence checking in reducing its work and the third option indicates to the checker that it can use the less expensive method that is applicable for only loop and data-reuse transformations. The last one provides the additional information required by the checker when certain properties have been assumed for algebraic transformations over specific operators appearing in the programs.

We have discussed that our program equivalence checking method requires that the original and the transformed programs belong to the allowed class of programs. However, the original and the transformed programs may not belong to the class of allowed programs to start with. In such a case, source code pre-processing tools are used in order to translate them to a form that is acceptable. We discuss this further in Section 3. Once the two programs have been pre-processed, we use an ADDG extractor that we have implemented in order to represent them as ADDGs. Central to this extractor is `per` [7, 8], a tool that provides various polyhedral domains used in constructing an ADDG. Our equivalence checker takes the two ADDGs as input and applies the method we discussed in Section 1.1. The checker either terminates with a successful completion of the traversal proving the two programs to be functionally input-output equivalent or produces diagnostic information in the case of a failure.

3 Pre-processing the Source Code

Typically, as can be expected in practice, the original and the transformed program pairs do not fall in the class of allowed programs that we have assumed for our method, at least not in all respects. But several crucial restrictions can be relaxed by using source code pre-processing tools. They are used to pre-process the initial and the transformed programs separately, before passing them to our equivalence checker. We use four specific source code pre-processors in our tool chain and they are user-demand driven. The sequence of tools in the chain are as shown in Figure 2. They are – (1) selective in-lining of functions in order to handle inter-procedural transformations (2) *if-conversion*, i.e., removal of any data-dependent control-flow; (3) conversion to *dynamic* single assignment form, i.e., removal of all false dependencies; and (4) DEF-USE checking that validates the schedule of reads and writes. Once a program is pre-processed, the true data dependencies between the variables and the operators in the program is represented as an *array data dependence graph* (ADDG). The constructed ADDGs of the two programs are input to the equivalence checker. In this section, we discuss each of the individual source code pre-processors.

3.1 Selective Function-Inlining

As discussed earlier, our checker functions intra-procedurally. But it is quite possible, for instance, that in the transformed program a program function has been inlined at its call-site and its internal loops exposed for transformations with other loops in the calling function. In order to handle such transformations,

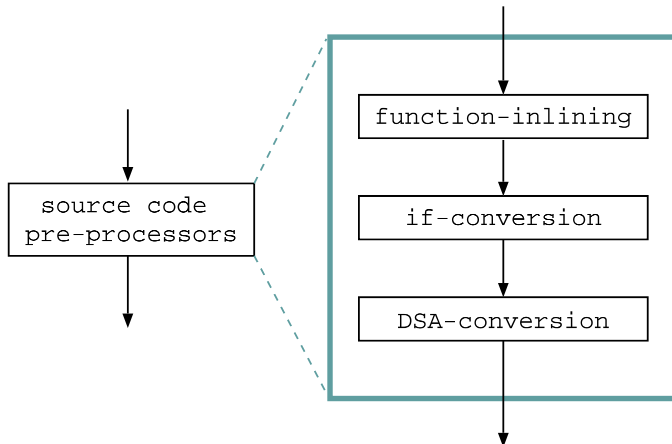

Figure 2: Chaining available code pre-processing tools.

we use a function inlining tool [1] that selectively inlines all functions that are called by the root function that provides scope for application of global loop and data-flow transformations.

3.2 If-Conversion

A data-dependent control-flow arises when there are `if`-statements or `while`-loops with references to variables in their conditional expressions[1]. It imposes additional dependence constraints on the execution of assignment statements in the program and hinders its analyses and transformations. This is particularly so for analyses that are primarily based on reasoning on the data-flow of the program. For such analyses, it is convenient if there is a possibility of encoding the control-flow as a data-flow. An example for such an encoding is the well known *if-conversion* [2] that removes data-dependent control dependencies in the program by converting them into data dependencies.

Our equivalence checking method is based on reasoning on the data dependencies and the ADDG representation that it uses is able to capture only data dependencies. Hence, it becomes necessary that a program is free from all data-dependent control dependencies before it can be represented as an ADDG. For every assignment statement in the program within the body of the `if`-statement, if-conversion introduces into the data-flow an `if`-operator. The operator has two operands, viz., (1) a conditional expression and (2) the righthand-side expression of the assignment statement. We use an if-conversion tool that has been developed in-house in order to achieve this [11]. Once if-conversion has been applied on a program, its representation as an ADDG is straightforward. The `if` operator is treated in the same way as any other operator.

The examples that follow show the basic cases of program codes with `if`-statements. Figure 3 shows a program function with a simple data-dependent `if`-statement on a single assignment statement and its ADDG representation.

When the `if`-statement also has an `else`-body, the assignment of values is controlled by the negation of the predicate in the condition of the `if`-statement. The natural candidate for representing an `if-then-else`-statement, is by adding another `if`-operator for the `else`-body with a logical negation operator inserted before the condition. Figure 4 shows an example program in this representation.

[1] Data-*independent* control-flow is based on conditional expressions on the iterators.

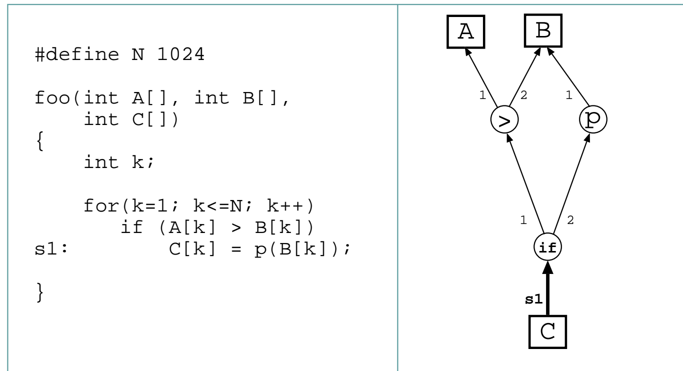

Figure 3: An example program function with a data-dependent `if`-statement and its ADDG obtained by if-conversion.

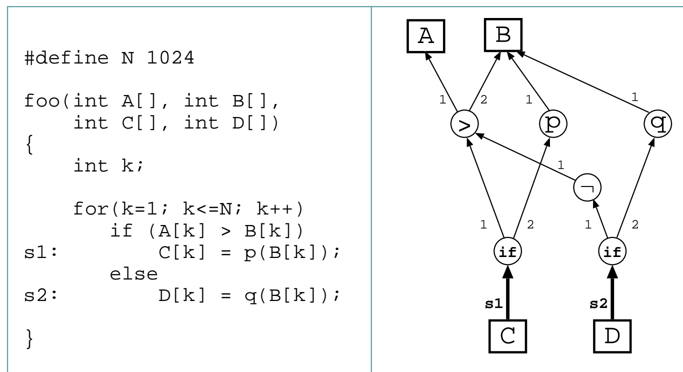

Figure 4: An example program function with a data-dependent `if-then-else`-statement and its ADDG obtained by if-conversion.

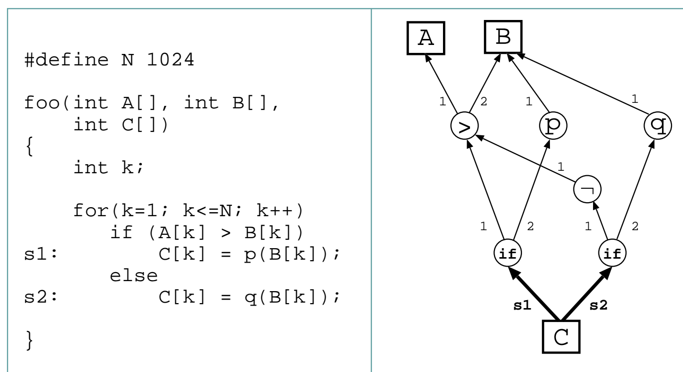

Figure 5: An example program function with a data-dependent `if-then-else`-statement and its ADDG obtained by if-conversion. Here, the statements `s1` and `s2` do not assign values to mutually disjunct domains of `C[]`.

The examples that follow in Figures 5-8 show some transformations on programs with data-dependent if-conditions and the effect they have on the ADDG representation. The equivalence checker with the knowledge of the algebraic properties of the logical operators invokes the flattening, some normalizing reductions and matching operations in identifying corresponding traversal paths.

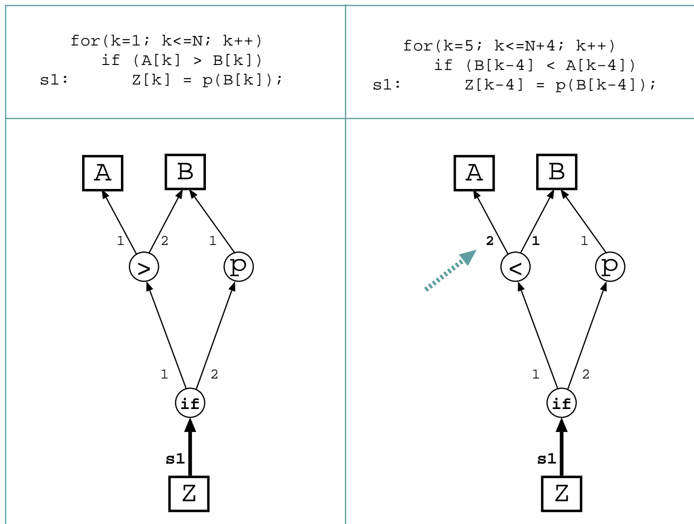

Figure 6: An example function pair where the relational operator < in the original has been replaced with its dual in the transformed program. In the ADDGs, the position labels on the operator account for the transformation.

3.3 DSA-Conversion

We require that input programs be in dynamic single assignment (DSA) form, that is, other than iterator variables, all variables in the program are written only once during program execution. When the original and the transformed programs are not in DSA form we first apply DSA-conversion to them. This is achieved by using a prototype tool that implements a generic and CPU-efficient (scalable to real-sized programs) method that is described in [16].

3.4 DEF-USE Checking

We assume that the input programs have a valid memory access schedule, that is, all reads from a memory location occur only after writing to the location. This helps our method by easing the verification, since commutativity of statements need no longer be checked. Before invoking our equivalence checker, we validate the assumption of a valid schedule, by using an independent DEF-USE checker, that is available in a tool-suite for application of loop transformations [18].

This helps in easing the verification, since commutativity of statements need no longer be checked. Once a program is pre-processed, the true data dependencies between the variables and the operators in the program is represented as an *array data dependence graph* (ADDG). The constructed ADDGs of the two programs are input to the equivalence checker.

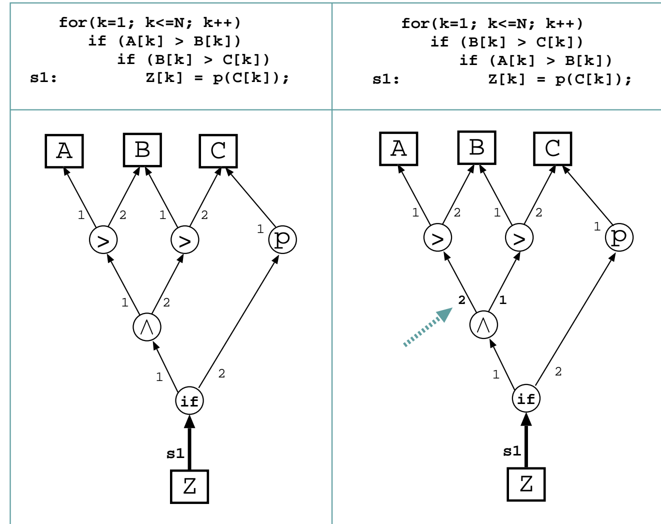

Figure 7: An example function pair where the if-conditions have been commuted. The $\wedge$-operator is commutative, therefore, matching-operation has to be invoked.

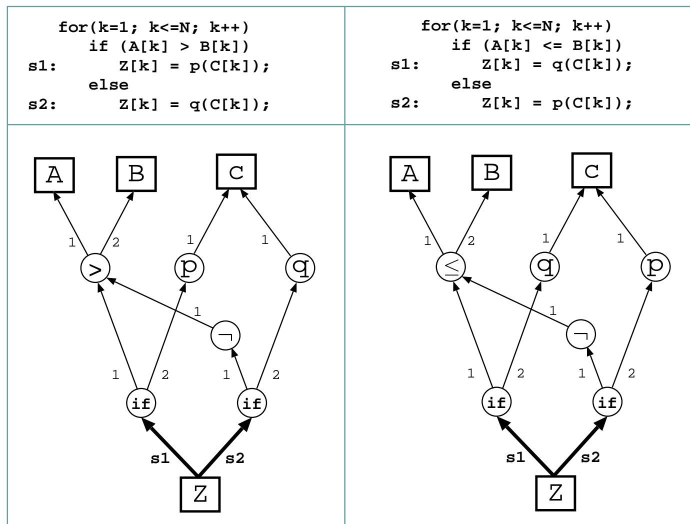

Figure 8: An example function pair where the bodies of the `then`-part and `else`-parts of the `if`-statement are swapped, with a corresponding replacement of the relational operator $>$ with its complementary operator $\leq$.

4 Case Studies

In this section, we report our experience with the current version of the equivalence checker tool on some kernels taken from actual applications in practice. In Section 4.1 we note the limitations of the tool, in Section 4.2 we compile the characteristics of the application programs that reflect their code complexity and in Section 4.3 we report the results of our experiments with the tool.

4.1 Implementation Characteristics

At present, we have implemented our formal equivalence checking methods in proof-of-concept prototype tools. Therefore, their capabilities are rather limited. Moreover, the formalization of the algorithms that we have discussed have evolved since the implementation of the prototype tool. In terms of functionality, it does not handle algebraic data-flow transformations and recurrence handling is limited. Also, it does not include the tabling mechanism and handling of the reconvergent paths. In terms of scalability, the implementation is able to handle modest size programs. Scalability to larger programs is however achievable by extending the techniques with better heuristics, for example, to handle recurrences. Future work will have to address this.

4.2 Application Characteristics

The code kernels that we have used from different applications and representative measures of their complexity is as shown in Tables 1 and 2. Note that in the case of GaussBlur and USVD, we have increased the dimensions of the array variables and created additional versions to check their impact on the time required for verification.

4.3 Verification Characteristics

The verification of the original and the transformed versions of the programs required times in the order of a few seconds as show in Tables 3 and 4. It also includes versions of the transformed code with errors introduced in them. As can be noted, the time required for verification does not degrade in the presence of errors. Also, as shown by verification of versions with higher dimensions of arrays, the impact on the time required for verification is negligibly small.

5 Summary

In this report, we have discussed the code pre-processing tools that are required in order to use our method in practice. With some modest experiments we have shown the feasibility of the method when applied to code kernels taken from some representative applications. However, further work is required to address some of the issues related to the implementation of our method in order to scale it to larger applications.

References

[1] M. J. Absar, P. Marchal, and F. Catthoor. Data-access optimization of embedded systems through selective inlining transformation. In *3rd Workshop on Embedded Systems for Real-time Multimedia (ESTIMedia)*, pages 75–80. IEEE, 2005.

Legend:

v = Code version,
l = Number of lines of uncommented code,
s = Number of assignment statements,
p = Number of statement classes,
q = Range of sizes of statement classes,
i = Number of input variables,
t = Number of intermediate variables,
o = Number of output variables,
n = Number of loop nests,
d = Depth of the ADDG,
r = Dimensions of array variables.

Application	v	l	s	p	q	n	r
USVD	1	450	310	24	4-25	46	3
	2	202	67	24	1-7	16	3
	3	450	310	24	4-25	46	6
	4	202	67	24	1-7	16	6

Table 1: Measures of complexity in application code fragments for which we have used the statement-level equivalence checking method.

Application	v	l	s	i	t	o	n	d	r
LUD	1	23	2	1	0	1	1	4	2,3
	2	48	16	1	13	1	2	7	2,3
Durbin	1	63	8	4	0	4	7	7	2
	2	56	8	4	0	4	8	7	2
M4ME	1	56	6	4	0	2	3	3	2,4,6
	2	62	8	4	1	2	4	4	2,4,5,6
GaussBlur	1	65	8	2	5	1	5	10	2,3
	2	62	8	2	5	1	5	10	2,3
	3	62	8	2	5	1	5	10	2,3,8

Table 2: Measures of complexity in application code fragments for which we have used the operator-level equivalence checking method.

Legend:

o	=	Version of the original code fragment,
t	=	Version of the transformed code fragment,
Wall	=	Wall clock time taken to prove equivalence in seconds,
CPU	=	CPU clock time taken to prove equivalence in seconds.

Application	o	t	Wall	CPU
USVD	1	2	19.54	0.17
USVD erroneous	1	2	19.11	0.13
USVD	3	4	19.85	0.15

Table 3: Measures of verification complexity with statement-level equivalence checking method.

Application	o	t	Wall	CPU
LUD	1	2	4.99	0.27
Durbin	1	2	8.57	0.39
Durbin: erroneous 1	1	2	8.51	0.34
Durbin: erroneous 2	1	2	5.32	0.21
M4ME	1	2	12.75	0.44
M4ME: erroneous	1	2	11.83	0.41
GaussBlur	1	2	12.27	0.69
GaussBlur	1	3	12.46	0.72

Table 4: Measures of verification complexity with operator-level equivalence checking method.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren. Conversion of control dependence to data dependence. In *Symposium on Principles of Programming Languages*, pages 177–189. ACM, 1983.

[3] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. The impact of source code transformations on software power and energy consumption. *Journal of Circuits, Systems, and Computers*, 11(5):477–502, 2002.

[4] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. V. Achteren, and T. Omnès. *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Academic Publishers, 2002.

[5] F. Catthoor, S. Wuytack, E. de Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[7] E. De Greef. *Storage Size Reduction for Multimedia Applications*. PhD thesis, Departement Elektrotechniek, Katholieke Universiteit Leuven, Belgium, 1998.

[8] E. De Greef. `per` - polyhedral extraction routines, 1998-2005.

[9] H. Falk and P. Marwedel. *Source Code Optimization Techniques for Data Flow Dominated Embedded Software*. Kluwer Academic Publishers, 2004.

[10] P. Feautrier. Array expansion. In *International Conference on Supercomputing*, pages 429–441. ACM, 1988.

[11] M. Palkovic, E. Brockmeyer, P. Vanbroekhoven, H. Corporaal, and F. Catthoor. Systematic preprocessing of data dependent constructs for embedded systems. In *International Workshop on Integrated Circuit and System Design, Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Lecture Notes in Computer Science (LNCS), pages 89–98. Springer, 2005.

[12] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.

[13] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In *Design, Automation and Test in Europe (DATE)*, pages 1310–1315. IEEE Computer Society, 2005.

[14] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Verification of source code transformations by program equivalence checking. In R. Bodík, editor, *International Conference on Compiler Construction*, volume 3443 of *Lecture Notes in Computer Science (LNCS)*, pages 221–236. Springer, 2005.

[15] R. A. van Engelen and K. A. Gallivan. An efficient algorithm for pointer-to-array access conversion for compiling and optimizing DSP applications. In *International Workshop on Innovative Architectures for Future Generation High-Performance Processors and Systems*, pages 80–89. IEEE, 2001.

[16] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, and F. Catthoor. A practical dynamic single assignment transformation. *ACM Trans. Design Autom. Electr. Syst.*, 12(4):40:1–40:21, 2007.

[17] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor. A step towards a scalable dynamic single assignment conversion. Technical Report CW 360, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2003.

[18] S. Verdoolaege. *Incremental Loop Transformations and Enumeration of Parametric Sets*. PhD thesis, Departement Computerwetenschappen, Katholieke Universiteit Leuven, Belgium, 2005.

[19] W. Wolf and M. Kandemir. Memory system optimization of embedded software. *Proceedings of the IEEE*, 91(1):165–182, 2003.