Efficiently Building the Gated Single Assignment Form in Codes with Pointers in Modern Optimizing Compilers[⋆]

Manuel Arenaz, Pedro Amoedo, and Juan Touriño

Computer Architecture Group
Department of Electronics and Systems
University of A Coruña, A Coruña, Spain
{arenaz,pamoedo,juan}@udc.es

Abstract. Understanding program behavior is at the foundation of program optimization. Techniques for automatic recognition of program constructs characterize the behavior of code fragments, providing compilers with valuable information to guide code optimizations. The XARK compiler framework provides a complete, robust and extensible solution to the automatic recognition problem that was shown to be effective to characterize the behavior of Fortran77 applications. Our goal is to migrate XARK to the GNU GCC compiler in order to widen its scope of application to program constructs (e.g., pointers, objects) supported by other programming languages (e.g., Fortran90/95, C/C++, Java). The first step towards this goal is the translation of the GCC intermediate representation into the Gated Single Assignment (GSA) form, an extension of Static Single Assignment (SSA) that captures data/control dependences and reaching definition information for scalar and array variables. This paper presents a simple and fast GSA construction algorithm that takes advantage of the infrastructure for building the SSA form available in modern optimizing compilers. An implementation on top of the GIMPLE-SSA intermediate representation of GCC is described and evaluated in terms of memory consumption and execution time using the UTDSP, Perfect Club and SPEC CPU2000 benchmark suites.

1 Introduction

Automatic code optimization hinges on advanced symbolic analysis to gather information about the behavior of programs. Compiler techniques for automatic kernel recognition carry out symbolic analysis in order to discover program constructs that are frequently used by software developers (e.g., inductions, scalar

[⋆] This research was supported by the Ministry of Education and Science of Spain and FEDER funds of the European Union (Projects TIN2004-07797-C02 and TIN2007-67537-C03), and by the Galician Government (Projects PGIDIT05PXIC10504PN and PGIDIT06PXIB105228PR). We gratefully thank the ArTeCS group (Universidad Complutense de Madrid, Spain) for the profile information of the SPEC2000 benchmarks.

reductions, irregular reductions and array recurrences). XARK [7] (first presented in [4]) is an extensible compiler framework for the recognition of a comprehensive collection of kernels that appear in real codes with regular and irregular computations, even in the presence of complex control flows. The framework analyzes the Gated Single Assignment (GSA) form [12] in order to handle data and control dependences, as well as scalar and array variables in a unified manner. XARK was shown to be an effective tool to support parallel code generation [5], compile-time prediction of the cache behavior [3], and program behavior characterization [6].

The current implementation of XARK is built on top of the Polaris compiler [13], which limits its scope of application to the analysis of Fortran77 codes. In order to handle codes written in other programming languages such as Fortran90/95, C/C++ and Java, XARK is being ported to the GIMPLE intermediate representation of the GNU GCC compiler [1]. Advantages of using GCC as a research and development compiler platform is that it is supported by an increasing number of industrial and academic institutions, and that GCC is able to compile codes written in many programming languages for a wide range of computer architectures.

The contribution of this paper is a simple and fast algorithm for the construction of the GSA form taking advantage of the infrastructure for building the Static Single Assignment (SSA) form [9] available in modern optimizing compilers. The key idea is to create artificial scalars that represent the memory regions accessed through both array references and pointer dereferences, and later handle these artificial scalars with the underlying SSA infrastructure. An advantage of this approach is that it enables an optimizing compiler to run GSA-based program transformations in the scope of an SSA-driven compilation process. An implementation in the scope of the GIMPLE-SSA intermediate representation of GCC is presented and evaluated with well-known benchmarks from different application domains, namely, UTDSP, Perfect Club and SPEC CPU2000.

The rest of the paper is organized as follows. Section 2 describes the GSA form and introduces the GSA construction algorithm. Section 3 describes the implementation in the scope of GIMPLE-SSA. Section 4 shows detailed experiments that show the efficiency of the implementation. Finally, Section 5 concludes the paper and outlines future work.

2 Algorithm for the Construction of the GSA Form

Modern optimizing compilers use intermediate representations based on the SSA form in order to facilitate certain code optimizations. GSA is an extension of SSA where the special ϕ operators inserted in the code capture both the reaching definitions of scalar and array variables, and the predicates of the conditional statements of the program. In GSA, different kinds of ϕs are distinguished:

- $\mu(x_{out}, x_{in})$, which appears at loop headers and selects the initial x_{out} and loop-carried x_{in} values of a variable.

```
procedure build_GSA()
{
 1. Build the SSA form
 2. Create pseudoscalars for each array/pointer variable
 3. Replace array references and pointer dereferences with different
    versions of the pseudoscalars
    Replace the statements defining pseudoscalars with $\alpha$ operators
 4. Delete pseudoscalars of read-only variables
 5. Foreach pseudoscalar
     {
       5.1. Run the SSA $\phi$-placement algorithm in order to insert $\phi$s
             at the confluence nodes of the Control Flow Graph
       5.2. Run the SSA variable renaming algorithm
     }
 6. Identify $\phi$ types at the confluence nodes of the Control Flow Graph
 7. Create predicates for the $\gamma$ operators
}
```

Fig. 1. Pseudocode of the algorithm for building the GSA form using the infrastructure for the construction of SSA form.

- $\gamma(c, x_{false}, x_{true})$, which is located at the confluence node associated with a branch (e.g., if-then-else construct), and captures the condition c for each definition to reach the confluence node: x_{false} if c is not fulfilled; x_{true}, if c is satisfied.
- $\alpha(a_{prev}, s, e)$, which replaces array assignment statements located within basic blocks, and whose semantics is that the s-th element of an array variable a is set to the value e, and the other elements take the values of the previous definition of the array (denoted as a_{prev}).

In general, building the GSA form involves three main tasks: (1) *ϕ-placement* for the insertion of μs and γs in the points of the program where the control flow merges; (2) *α-placement*, to replace the non-scalar assignment statements (e.g., arrays, pointer dereferences) with αs; and (3) *variable renaming* in order to assure that the left-hand sides of the assignment statements are pairwise disjoint. The key idea that enables the construction of the GSA form by means of the ϕ-placement and variable renaming algorithms of an existing SSA infrastructure is to replace array references and pointer dereferences with artificial scalar variables (from now on *pseudoscalars*). These variables represent the memory regions that can be accessed through arrays and pointers as a unique entity.

Figure 1 presents the GSA construction algorithm proposed in this paper. The pseudocode consists of seven steps. First, the program is rewritten into SSA form. Second, the symbol table is analyzed in order to create a pseudoscalar for each array variable and each pointer variable declared in the source code. Third, the SSA form is scanned in order to replace each array reference and pointer dereference with a new version of the corresponding pseudoscalar. Whenever a pseudoscalar is inserted in the left-hand side of a statement, the right-hand side is also replaced with an α operator (α-placement). Fourth, the pseudoscalars that represent read-only variables are deleted as they will not lead to the insertion of new ϕ operators in the fifth step. The fifth step uses the ϕ-placement and variable renaming algorithms provided by the SSA infrastructure to update the SSA form

```
pB = &B[0];
i = 0;
for (h=0; h<N; h++) {
   if (A[h] != 0) {
      *pB++ = A[h];
      C[i] = h;
      i++;
   }
}
```

Fig. 2. Gather operation implemented with array references and pointer dereferences.

with the reaching definitions of the pseudoscalars. Next, the sixth step classifies ϕ operators into μ or γ operators. Finally, for each γ, the control flow of the program that determines the reaching definitions of the γ operator is captured using the algorithm proposed in [10]. As a whole, this algorithm computes a directed acyclic graph that reflects the hierarchy of if-then-else constructs and that computes the set of predicates associated with each reaching definition of the γ operator.

3 Implementation using the GIMPLE-SSA Infrastructure

The algorithm described in the previous section has been implemented in the scope of the GIMPLE-SSA intermediate representation of the GNU GCC compiler. GIMPLE is a three-address language with no high-level control flow structures (e.g., loops, if-then-else), which are lowered to conditional gotos. Each GIMPLE statement contains no more than three operands and has no implicit side effects. Temporaries are used to hold intermediate values as necessary. Variables that need to live in memory are never used in expressions. They are first loaded into a temporary and the temporary is then used in the expression.

For illustrative purposes, Figure 2 shows an implementation of a gather operation that filters out the elements of an input array `A` that are equal to zero. Non-zero elements are stored in consecutive entries of an output array `B` by dereferencing the pointer `pB`. In addition, the indices of the non-zero elements are stored in the corresponding entries of the output array `C`. Figure 3(a) shows the GIMPLE-SSA form built by GCC. The beginning of basic blocks is labeled in the figure (e.g., $\texttt{<BB}_0\texttt{>}$). The process of building GIMPLE works recursively, replacing source code statements with sequences of GIMPLE statements. Thus, the source code statement `*pB++=A[h]` in basic block $\texttt{<BB}_2\texttt{>}$ is substituted by five GIMPLE statements that compute: (1) the offset of the element `h` with respect to the base address of the array `A` ($\texttt{D1}_1\texttt{=h}_1\texttt{*4}$); (2) the address of the memory location that contains the entry `A[h]` ($\texttt{D2}_1\texttt{=A+D1}_1$); (3) the value of the array element ($\texttt{D3}_1\texttt{=*D2}_1$); (4) the write operation to store that value in the entry of array `B` pointed by `pB` ($\texttt{*pB=D3}_1$); and (5) the increment of `pB` to point to the next entry of array `B` ($\texttt{pB}_2\texttt{=pB}_1\texttt{+4}$). Figure 3(a) also shows the variables declared in the program, and indicates the versions of each variable created during the construction of the SSA form.

```
/* Declarations */
pB :: pB_0,pB_1,pB_2,pB_3
A
B
C
i :: i_0,i_1,i_2,i_3
h :: h_0,h_1,h_2,h_3
D1 :: D1_0,D1_1
D2 :: D2_0,D2_1
D3 :: D3_0,D3_1
D4 :: D4_0
D5 :: D5_0

/* Statements */
<BB_0>:
  pB_0 = B;
  i_0 = 0;
  h_0 = 0;
  goto <BB_4>;

<BB_1>:
  D1_0 = h_1 * 4;
  D2_0 = A + D1_0;
  D3_0 = *D2_0;
  if (D3_0 != 0)
    goto <BB_2>
  else
    goto <BB_3>

<BB_2>:
  D1_1 = h_1 * 4;
  D2_1 = A + D1_1;
  D3_1 = *D2_1;
  *pB = D3_1;
  pB_2 = pB_1+4;
  D4_0 = i_1 * 4;
  D5_0 = C + D4_0;
  *D5_0 = h_1;
  i_2 = i_1+1;

<BB_3>:
  i_3 = φ(i_1,i_2);
  pB_3 = φ(pB_1,pB_2);
  h_2 = h_1+1;

<BB_4>:
  i_1 = φ(i_0,i_3);
  h_1 = φ(h_0,h_2);
  pB_1 = φ(pB_0,pB_3);
  if (h_1<=N) goto <BB_1>;
```

(a) After step 1.

```
/* Declarations */
pB :: pB_0,pB_1,pB_2,pB_3
A
B
C
i :: i_0,i_1,i_2,i_3
h :: h_0,h_1,h_2,h_3
D1 :: D1_0,D1_1
D2 :: D2_0,D2_1
D3 :: D3_0,D3_1
D4 :: D4_0
D5 :: D5_0
θpB :: θpB_0, θpB_1
θA :: θA_0,θA_1
θB
θC :: θC_0,θC_1

/* Statements */
<BB_0>:
  pB_0 = B;
  i_0 = 0;
  h_0 = 0;
  goto <BB_4>;

<BB_1>:
  D1_0 = h_1 * 4;
  D2_0 = A + D1_0;
  D3_0 = θA_0;
  if (D3_0 != 0)
    goto <BB_2>
  else
    goto <BB_3>

<BB_2>:
  D1_1 = h_1 * 4;
  D2_1 = A + D1_1;
  D3_1 = θA_1;
  θpB_0 = α(θpB_1,0,D3_1);
  pB_2 = pB_1+4;
  D4_0 = i_1 * 4;
  θC_0 = α(θC_1,D4_0,h_1);
  i_2 = i_1+1;

<BB_3>:
  i_3 = φ(i_1,i_2);
  pB_3 = φ(pB_1,pB_2);
  h_2 = h_1+1;

<BB_4>:
  i_1 = φ(i_0,i_3);
  h_1 = φ(h_0,h_2);
  pB_1 = φ(pB_0,pB_3);
  if (h_1<=N) goto <BB_1>;
```

(b) After steps 2 and 3.

```
/* Declarations */
pB :: pB_0,pB_1,pB_2,pB_3
A
B
C
i :: i_0,i_1,i_2,i_3
h :: h_0,h_1,h_2,h_3
D1 :: D1_0,D1_1
D2 :: D2_0,D2_1
D3 :: D3_0,D3_1
D4 :: D4_0
D5 :: D5_0
θpB :: θpB_0,θpB_1,θpB_2,θpB_3
θC :: θC_0,θC_1,θC_2,θC_3

/* Statements */
<BB_0>:
  pB_0 = B;
  i_0 = 0;
  h_0 = 0;
  goto <BB_4>;

<BB_1>:
  D1_0 = h_1 * 4;
  D2_0 = A + D1_0;
  D3_0 = *D2_0;
  if (D3_0 != 0)
    goto <BB_2>
  else
    goto <BB_3>

<BB_2>:
  D1_1 = h_1 * 4;
  D2_1 = A + D1_1;
  D3_1 = *D2_1;
  θpB_2 = α(θpB_1,0,D3_1);
  pB_2 = pB_1+4;
  D4_0 = i_1 * 4;
  θC_2 = α(θC_1,D4_0,h_1);
  i_2 = i_1+1;

<BB_3>:
  i_3 = γ(D3_0!=0,i_1,i_2);
  pB_3 = γ(D3_0!=0,pB_1,pB_2);
  θpB_3 = γ(D3_0!=0,θpB_1,θpB_2);
  θC_3 = γ(D3_0!=0,θC_1,θC_2);
  h_2 = h_1+1;

<BB_4>:
  i_1 = μ(i_0,i_3);
  h_1 = μ(h_0,h_3);
  pB_1 = μ(pB_0,pB_3);
  θpB_1 = μ(θpB_0,θpB_3);
  θC_1 = μ(θC_0,θC_3);
  if (h_1<=N) goto <BB_1>;
```

(c) After steps 4 to 7.

Fig. 3. Construction of the GSA form on top of GIMPLE-SSA using the algorithm of Figure 1 for the case study of Figure 2.

Table 1. Representation of C and Fortran declarations and uses in GIMPLE form.

Language	Source code		GIMPLE form		
	Declaration	Use	Declaration	Use	Type
C	parameter void f(int A[])	A[i]	int *A	D=A+i*sizeof(int) *D	2
	parameter void f(int *A)	*(A+i)			
	local int *A	*A	int *A	*A	1
	parameter void f(int A[N][M])	A[i][j]	int[M] *D	D=A+i*sizeof(int) (*D)[j]	2
	local int A[N][M]	A[i][j]	int[N][M] A	A[i][j]	3
Fortran	parameter subroutine f(A) integer A(N)	A(i)	int[N] *A	(*A)[i-1]	3
	parameter subroutine f(A) integer A(*)	A(i)	int[] *A		
	parameter subroutine f(A) integer A(N,M)	A(i,j)	int[N*M] *A	(*A)[(j-1)*N+(i-1)]	3
	local integer A(N)	A(i)	int A[N]	A[i-1]	3
	local integer A(N,M)	A(i,j)	int[N*M] A	A[(j-1)*N+(i-1)]	3

The translation of the GIMPLE-SSA form into GSA starts by creating pseudoscalars for each array variable (`A`, `B` and `C`) and for each pointer variable (`pB`) declared in the source code of the program. In Figure 3(b), pseudoscalars are denoted by the name of the source code variable prefixed by the symbol θ (θ`A`, θ`B`, θ`C` and θ`pB`). GIMPLE is a common representation for source codes written in different programming languages, and thus array references and pointer dereferences can be represented in many different ways. Table 1 presents the GIMPLE representation of several declarations and uses in C and Fortran. Two scopes are considered: *parameter* for subroutine arguments, and *local* for subroutine local variables. The last column identifies the three types of representations distinguished in the third step of the algorithm of Figure 1 for the creation of new versions of the pseudoscalars:

1. Dereferences of source code pointers, which are substituted by a new version of the pseudoscalar of that pointer (see $\texttt{*pB=D3}_1$ and $\theta\texttt{pB}_0\texttt{=}\alpha\texttt{(}\theta\texttt{pB}_1\texttt{,0,D3}_1\texttt{)}$ in basic block $\texttt{<BB}_2\texttt{>}$ of Figures 3(a) and 3(b), respectively).
2. Dereferences of temporaries, replaced with a new version of the pseudoscalar associated with the corresponding source array or pointer variable (see $\texttt{*D5}_0\texttt{=h}_1$ and $\theta\texttt{C}_0\texttt{=}\alpha\texttt{(}\theta\texttt{C}_1\texttt{,D4}_0\texttt{,h}_1\texttt{)}$ in basic block $\texttt{<BB}_2\texttt{>}$).
3. Dereferences of array variables and array references, substituted by a new version of the pseudoscalar of the corresponding source array variable.

The third step of the algorithm is completed by replacing with α operators those GIMPLE-SSA statements whose left-hand side has been substituted by a version of a pseudoscalar. The arguments of the α operator are: a new version of the left-hand side pseudoscalar, the offset of the memory location whose value is written during the execution of the statement, and the value assigned to the memory location. The offset is determined as follows. For type one, the offset is zero because the statement writes in the memory location pointed by the source pointer variable (see $\texttt{*pB=D3}_1$ and $\theta\texttt{pB}_0\ \texttt{=}\ \alpha\texttt{(}\theta\texttt{pB}_1\texttt{,0,D3}_1\texttt{)}$ in Figures 3(a) and 3(b), respectively). For types two and three, the offset is determined by the subscripts of the GIMPLE array reference or by the offset added to the source pointer variable (see `*D5`$_0$`=h`$_1$ and θ`C`$_0$`=`α`(`θ`C`$_1$`,D4`$_0$`,h`$_1$`)`).

The fourth step deletes the pseudoscalars corresponding to read-only variables as they will not lead to the insertion of new ϕ operators. Thus, the implementation of the fifth step is straightforward and consists of running the ϕ-placement and variable renaming algorithms available in the GCC infrastructure. For illustrative purposes, see the versions of the pseudoscalar θ`A` inserted in basic blocks `<BB`$_1$`>` and `<BB`$_2$`>` of Figure 3(b) after step two, which have been removed from the GSA form of Figure 3(c).

Finally, the different types of ϕ operators inserted by the ϕ-placement algorithm are identified. Thus, ϕs located at loop headers are converted into μ operators, and ϕs inserted after if-then-else constructs are converted into γ operators. The GSA form is completed with the computation of the predicate that controls the execution of the if-then-else construct in GIMPLE-SSA form (see predicate `(D3`$_0$`!=0)` in Figure 3(c)).

4 Experimental Results

The performance of our GSA construction algorithm was evaluated with the UTDSP [11], Perfect Club [8] and SPEC CPU2000 [2] benchmarks. UTDSP provides routines written in different coding styles (pointer-based and array-based) that are representative of DSP applications (e.g., filters, FFT). The well-known Perfect Club and SPEC CPU2000 benchmarks consist of full-scale applications that have been used extensively in the literature. As a first step towards the analysis of these applications as a whole, the most costly routines in terms of execution time were selected as they would probably be the target of an execution-time-aware optimizing compiler.

In order to characterize the behavior of an application, XARK addresses the recognition of the computational kernels whose results are stored in both scalar and non-scalar variables (e.g., arrays, pointers). As SSA covers scalar variables, the effectiveness of the GSA construction algorithm is measured as the percentage of non-scalar variables (*#non_scalars*) converted into pseudoscalars (*#pseudos*) successfully. Table 2 shows that there are some Fortran routines in Perfect Club and SPEC CPU2000 where *#pseudos* is lower than *#non_scalars*, which means that the GSA form can only be built partially (the effectiveness is less than 100%). The reason for this is that variables (in particular, arrays) declared in Fortran common blocks are represented in GIMPLE as fields of structure data types. This situation can be handled by distinguishing new types of GIMPLE representations in the third step of our algorithm, which is work in progress. Finally, note that the GSA form is built successfully (the effectiveness is 100%) in all the C routines as well as in most of the Fortran routines of Perfect Club and SPEC CPU2000.

Figure 4 compares memory consumption and execution time of the GSA construction algorithm (in white) with respect to the SSA construction algorithm of GCC (in black), the routines being ordered by increasing value of memory

Table 2. Effectiveness of the GSA construction algorithm.

Benchmark	Application	Routine	Language	#non_scalars	#pseudos	Effectiveness
UTDSP	COMPRESS	dct_array	C	2	2	100%
		dct_ptr	C	2	2	100%
	FFT	fft_array	C	2	2	100%
		fft_ptr	C	4	4	100%
	FIR	fir_array	C	1	1	100%
		fir_ptr	C	1	1	100%
	IIR	iir_array	C	2	2	100%
		iir_ptr	C	2	2	100%
	LATNRM	latnrm_array	C	2	2	100%
		latnrm_ptr	C	2	2	100%
	LMSFIR	lmsfir_array	C	2	2	100%
		lmsfir_ptr	C	2	2	100%
Perfect Club	ADM	dctdxf	Fortran	1	1	100%
		radb4	Fortran	1	1	100%
		radf4	Fortran	1	1	100%
	DYFESM	matmul	Fortran	1	1	100%
	FLO52	dflux	Fortran	7	4	57%
		eflux	Fortran	2	0	0%
	MDG	cshift	Fortran	1	1	100%
	MG3D	cpass	Fortran	2	2	100%
		cpassm	Fortran	2	2	100%
	QCD	mult	Fortran	1	1	100%
		observ	Fortran	2	2	100%
	TRFD	olda	Fortran	7	7	100%
SPEC CPU2000	APPLU	blts	Fortran	2	2	100%
		buts	Fortran	3	3	100%
		jacld	Fortran	4	0	0%
		jacu	Fortran	3	0	0%
		rhs	Fortran	2	1	50%
	APSI	radbg	Fortran	2	2	100%
		radfg	Fortran	2	2	100%
	EQUAKE	smvp	C	3	3	100%
	SWIM	calc1	Fortran	4	0	0%
		calc2	Fortran	3	0	0%
		calc3	Fortran	6	0	0%

consumption of GIMPLE-SSA. Memory consumption is calculated as the sum of the sizes of the symbol table (including the variables declared in the source code as well as the corresponding versions of each variable), the forest of abstract syntax trees, the control flow graph and the data dependence graph. Execution times were measured on an Intel Core2 Duo at 2.4Ghz (4MB of cache and 2GB of RAM). Overall, the results show that our algorithm consumes, on average, between 4% and 11% more memory than the SSA construction algorithm and takes between 14% and 31% longer. The memory overhead varies between 3% and 21%, and the time overhead between 4% and 43%. However, while the time overhead may seem significant, the execution time does not exceed 5ms in the worst case. Thus, the experiments demonstrate that the algorithm is a practical solution for optimizing compilers that require advanced program analysis techniques.

5 Conclusions

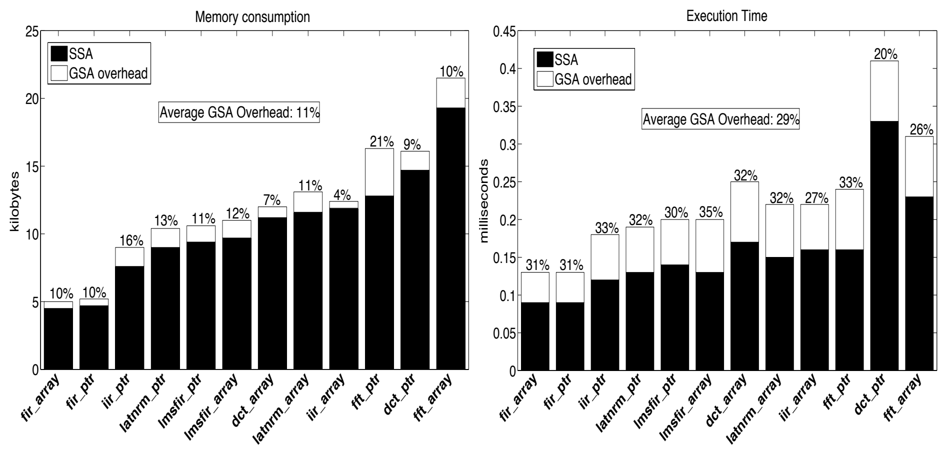
This paper has presented a simple algorithm for building the GSA representation using the infrastructure for building SSA available in modern optimizing compilers. The approach based on the concept of pseudoscalar as a representation of references to array and pointer variables was shown to be effective for the analysis of C and Fortran codes from different application domains.

The algorithm has been evaluated in terms of memory consumption and execution time. The experiments have shown that the algorithm introduces an affordable overhead to the underlying SSA implementation. Thus, it provides a practical solution that enables the coexistence of SSA- and GSA-based optimizations. In addition, it provides support for the efficient implementation of advanced analysis techniques targeting pointer- and array-based codes in the scope of widely extended compiler platforms such as GCC.

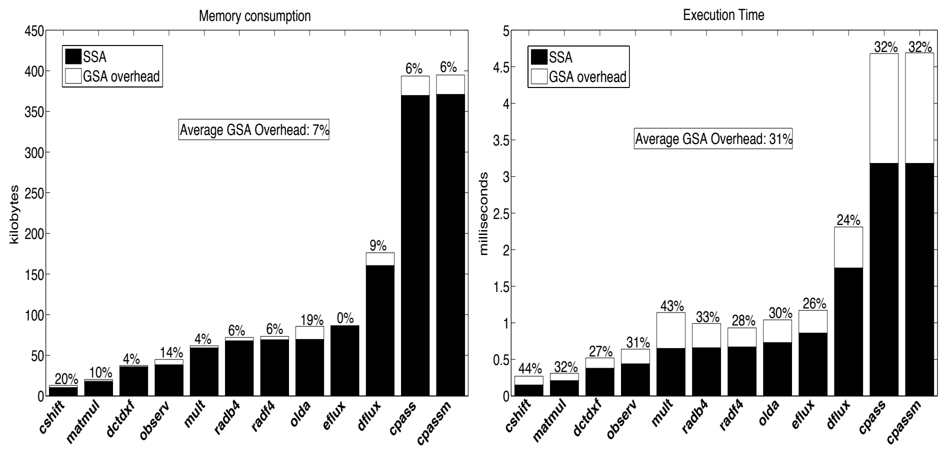
As future work we intend to analyze applications written in object-oriented programming languages like C++ and Java.

References

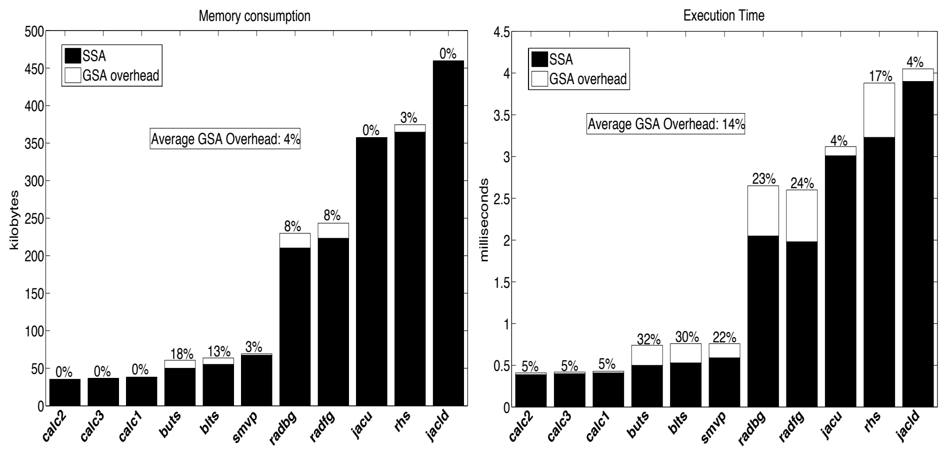
1. GNU Compiler Collection (GCC) internals. Available at `http://gcc.gnu.org/onlinedocs/gccint.pdf` [Last accessed June 2008].
2. Standard Performance Evaluation Corporation. SPEC CPU2000. Available at `http://www.spec.org/cpu2000` [Last accessed June 2008].
3. Andrade, D., Arenaz, M., Fraguela, B.B., Touriño, J., Doallo, R. Automated and accurate cache behavior analysis for codes with irregular access patterns. *Concurrency and Computation: Practice and Experience*, 19(18):2407–2423 (2007)
4. Arenaz, M., Touriño, J., Doallo, R. A GSA-based compiler infrastructure to extract parallelism from complex loops. In: 17th ACM International Conference on Supercomputing, San Francisco, CA, pp. 193–204 (2003)
5. Arenaz, M., Touriño, J., Doallo, R. Compiler support for parallel code generation through kernel recognition. In: 18th IEEE International Parallel and Distributed Processing Symposium, Santa Fe, NM (2004)
6. Arenaz, M., Touriño, J., Doallo, R. Program behavior characterization through advanced kernel recognition. In: 13th International Euro-Par Conference, Rennes, France, pp. 237–247 (2007)
7. Arenaz, M., Touriño, J., Doallo, R. XARK: An eXtensible framework for Automatic Recognition of computational Kernels. *ACM Trans. Program. Lang. Syst.* (accepted for publication)
8. Berry, M., et al. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. *Int. J. Supercomputer Apps.*, 3(3):5–40 (1989)
9. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490 (1991)
10. Havlak, P. Construction of thinned gated single-assignment form. In: 6th International Workshop on Languages and Compilers for Parallel Computing, Portland, OR, pp. 477–499 (1993)
11. Lee, C.G. UTDSP benchmarks. Available at `http://www.eecg.toronto.edu/-/~corinna/DSP/infrastructure/UTDSP.html` [Last accessed June 2008].
12. Tu, P., Padua, D.A. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In: 9th ACM International Conference on Supercomputing, Barcelona, Spain, pp. 414–423 (1995)
13. Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D.A., Paek, Y., Pottenger, W.M., Rauchwerger, L., Tu, P. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82 (1996)



(a) UTDSP.



(b) Perfect Club.



(c) SPEC CPU2000.

Fig. 4. Memory consumption and execution time of the GSA construction algorithm.