Using Software Metrics to Evaluate Static Single Assignment Form in GCC

Jeremy Singer[1], Christos Tjortjis[2], and Martin Ward[3]

[1] School of Computer Science, University of Manchester, UK
jsinger@cs.man.ac.uk
[2] Department of Computer Science, University of Ioannina, Greece
[3] Software Technology Research Laboratory, De Montfort University, UK

Abstract. Over the past 20 years, static single assignment form (SSA) has risen to become the compiler intermediate representation of choice. Compiler developers cite many qualitative reasons for choosing SSA. However in this study, we present clear *quantitative* benefits of SSA, by applying several standard software metrics to compiler intermediate code in both SSA and non-SSA forms. The average complexity reduction achieved by using SSA in the GCC compiler is between 32% and 60% according to our software metrics, over a set of standard SPEC benchmarks.

1 Introduction

Static single assignment form (SSA) is a popular compiler IR. Since the formulation of SSA in the late 1980s [9] it has rapidly become the de factor standard IR for code analysis and optimization.

In terms of research compiler infrastructures, SUIF [17], Microsoft's Bartok [11] and Phoenix, IBM's Jikes RVM [6], and LLVM all use SSA-based intermediate forms. Several commercial compilers have recently been released as open-source. Among these, Java HotSpot (originally from Sun) and Open64 (originally from SGI) use SSA. Although Intel's C compiler is not open-source, it has been reported to make heavy use of SSA-based optimizations [33]. Regarding the open-source GNU Compiler Collection (GCC), its development began before SSA was well-characterized or widely known. However SSA support has been backported into the original optimization infrastructure [30, 31], as of version 3.4. Thus we see that many compilers, from a wide variety of vendors, for a diverse range of source programming languages and target architectures, use SSA.

Despite its popularity and many anecdotal success stories, there has been little previous work on formal evaluation of the reasons for SSA's advantages in static analysis and optimization. This paper employs software metrics to provide a *quantitative* evaluation of the comparative merits of SSA. We aim to make our observations as general as possible, by avoiding any restrictions or assumptions that are specific for systems, analyses or optimizations.

The major contribution of this paper is in the previously unchartered area of *program meta-analysis*, which concerns the analysis of properties of program

analysis tools and techniques. The paper provides a *quantitative assessment* of the benefits of SSA in an analysis-independent manner, by means of *software metrics*. It shows with several standard measures that an SSA representation of a procedure has reduced complexity in relation to a non-SSA representation of the same procedure, in GCC. The average *complexity reduction* is between **32% and 60%** for selected complexity metrics, over a set of representative benchmark programs.

2 Program Representations

A control flow graph (CFG) is a static representation of possible flows of execution in a procedure. The CFG form is the basis for almost all classical (pre-SSA) intra-procedural data flow analysis techniques. For full details, consult compiler textbooks [1] [27] [29].

The conversion from CFG to SSA involves *systematic renaming* of variables. The key property of SSA is that each variable in the program must have a unique (hence *single*) definition point (hence *assignment*) in the program text (hence *static*). In order to achieve this, it is necessary to generate new variable names at definition points, and propagate these new names to all uses reached by that definition. At control-flow merge points reached by several definitions of a variable, it may be necessary to insert pseudo-assignments (known as ϕ-functions) to merge explicitly these renamed variable definitions into a new variable. In SSA form, each unique variable definition must dominate all uses of that variable.

SSA is perceived to improve data flow analysis, since: (1) it decreases the size of def-use chains; and (2) it enables finer-grained variable-specific data flow information.

A *def-use* chain links a variable definition with all its uses. It is an additional data structure alongside a CFG, encapsulating variable data flow in the program. For instance, if there are 10 definitions of variable x which can reach a control flow merge point, followed by 10 uses of x, then every definition can reach every use. This means there are 100 data flow links for x. In the SSA version of the program, there are just 20 data flow links: 10 from the definitions to the ϕ-function at the merge point, and 10 from the ϕ-function to the uses. Thus SSA can be seen to simplify def-use chains. Many authors extol this virtue of SSA and its consequent effect on splitting live ranges [10, 5, 19, 28, 4].

SSA enables effective sparse data flow analysis: Data flow information can be associated with a variable globally, rather than at each specific control flow point in the program [37]. Hasti and Horwitz assert that SSA encodes control flow information directly into variable names, which means that flow-insensitive analysis of SSA is as accurate as flow-sensitive analysis in certain cases [13].

Thus it is commonly recognised that SSA gives some advantages over the CFG. Research papers on SSA generally provide comparative evaluations of CFG *versus* SSA for a single data flow problem, compiler optimization or system; for example [2]. However as far as we are aware, the current paper is the *first*

to address the general issue of why SSA is better than CFG in generic and quantitative terms.

3 Software Metrics

This section describes the various metrics we use, and how we adapt these metric definitions for SSA. Our objective is to use software metrics to measure the differences between CFG and SSA representations of a program. So this means that absolute metric values are not important; we are only concerned with the relative values for CFG and SSA representations of same program.

There are two novelties to our approach:

1. We apply software metrics to auto-generated code rather than to human-generated code. This is not common practice, although it has been done previously for specialized analyses [36, 35, 24].
2. We apply software metrics to compiler IR code rather than to high-level program source code. To the best of our knowledge, this is the first instance of such an application. We expect this is due to the limited interaction between the software metrics and program analysis research communities.

This section focuses on *intra-procedural* metrics. SSA and CFG are intra-procedural representations. Although inter-procedural versions exist [22, 34] they are not in widespread use.

Since the CFG and SSA representations of a procedure share the same control flow structure, we cannot use control flow metrics like cyclomatic complexity [23] to compare them. Instead, we must concentrate on metrics that depend on *variable naming* conventions. Note that we are considering compiler variables (sometimes known as temporaries or virtual registers), rather than source code variables.

3.1 Size Metric

We measure a procedure's size by the *number of basic blocks* in the graph, i.e. $|B|$ for a CFG $(B, E, b_{\text{entry}}, b_{\text{exit}})$. This size metric is more appropriate for graph-based IRs than the most common metric: source lines of code. Note that a procedure will have the *same size* in both SSA and non-SSA forms.

In SSA program analysis, computational complexity measures are generally size related. For a program of size n, the time complexity of the standard SSA transformation algorithm is $O(n^2)$. The SSA vocabulary space complexity and number of ϕ-functions are also $O(n^2)$.

3.2 Halstead Metrics

Halstead's complexity metrics [12] were originally developed to measure the complexity of program modules directly by analysis of source code. They are among the earliest software metrics. The Halstead measures are based on four integer values:

1. $n1$—the number of distinct operators
2. $n2$—the number of distinct operands
3. $N1$—the total number of operators
4. $N2$—the total number of operands

In our case, we take *operands* to be virtual registers in the compiler IR code, and *operators* to be instructions and pseudo-instructions. Appendix A gives a full description of how we interpret operators and operands in our metric calculations.

From these four basic values, five complexity metrics are derived, as shown in Table 1.

Table 1. Table of Halstead Complexity Metrics

Measure	Formula
Program length	$N = N1 + N2$
Program vocabulary	$n = n1 + n2$
Volume	$V = N * (\log_2 n)$
Difficulty	$D = (n1/2) * (N2/n2)$
Effort	$E = D * V$

Measurements $n1$ and $N1$ concern number of operators. These numbers will be identical for CFG and SSA programs if we exclude ϕ-functions from consideration.[4] Therefore the only differences occur with $n2$ and $N2$, which concern the names of operands. $n2$ should change from CFG to SSA, since it measures number of distinct operands. On the other hand, $N2$ should remain unchanged, since SSA only renames *existing* operands and does not introduce new instructions. (Again, we exclude ϕ-functions from consideration.) Generally $n2$ will be greater for SSA than CFG, which means that SSA volume V should be larger than CFG, and SSA difficulty D should be smaller than CFG. Since the volume increase is logarithmic, whereas the difficulty decrease is linear, the overall SSA effort E should also decrease in relation to CFG effort.

A high effort score is undesirable; it means that a program module is difficult to understand and maintain. Section 5.1 presents our empirical results for analysis using these Halstead metrics.

3.3 Information Flow Complexity Metric

Henry and Kafura present the information flow complexity metric (IFC) [15] as a *system-level* design metric. It provides a measure of the information that flows between the various modules in a system. In the original study [15] it was applied

[4] We consider this to be reasonable, since ϕ-functions are only copies rather than real computational operations. A simple copy instruction does not count as an operator, so neither does a ϕ-function.

to source code modules in UNIX. IFC has since been used for system specification studies [18] and iterative software design [16]. IFC is often used at a lower level than system modules; it is very commonly used to quantify information flowing between source-level procedures in a program [21]. IFC was deemed to be a measure that could be used to 'produce reliable software' enshrined in IEEE Standard 982.2.

IFC depends on the amount of information flowing into a module, known as fanIn, through parameters and reads of global data structures. IFC also depends on the amount of information flowing out of a module, known as fanOut, through return values and writes to global data structures. Finally IFC depends on the length of the module. The original formula for calculating the IFC for a module is given in equation 1.

$$\text{IFC} = \text{length} * (\text{fanIn} * \text{fanOut})^2 \tag{1}$$

A high IFC score for a module is undesirable. Long modules, and those involved in lots of information flows have high IFC. The message of the original paper is that high IFC indicates *lack of maintainability*. Lots of cross-module dependencies make it difficult to change the system. This is similar to the more recent notion of *coupling* in object-oriented software development [7].

In this study, we adopt IFC to measure the complexity of individual basic blocks, at an intra-procedural level. Although it does not seem that Henry and Kafura's Information Flow metric is immediately applicable here, we re-interpret the SSA representation (rather than modify the metric) to make the application straightforward.

Procedural information flow relies on measuring number of *input* and *output* variables in each procedure. This simply equates to parameters as inputs and return values as outputs in most high-level languages. A single procedure in SSA can be viewed as a collection of smaller functions in a call graph, where each basic block from the original procedure now corresponds to a function in the call graph. Kelsey [20] and Appel[3] give the details of this transformation. They argue that SSA is a form of *functional programming*. Each basic block from the original procedure now becomes a *function*. Upwardly exposed uses (variables used in a basic block that are not previously defined in that same block) become input parameters. Variables whose definitions in this basic block reach to other blocks (variables that live on exit from this basic block) become output parameters. We transform our SSA procedures into these kinds of functional programs, then apply the IFC metric to each function.

Recalling the definition of IFC from equation 1, we now need to clarify how to evaluate it on a given function.

- The length is the number of operations in the function, which should correspond to the instructions in the original basic block.
- The fanIn is the number of input parameters for the function, which should correspond to the number of upwardly exposed variable uses in the original basic block.

– The fanOut is the number of output parameters for the function, which should correspond to the number of variables live on exit from the original basic block.

We aim to compute IFC scores for basic blocks in CFG and SSA IRs, and compare them. To make this comparison, we analyse functions with their SSA variable names to measure fanIn and fanOut, and to compute IFC. Then we remove subscripts from the SSA variable names to recover an approximation to the original CFG variable names. Then we measure fanIn and fanOut for these CFG variables, and compute IFC scores for the functions with CFG variable names. Section 5.2 presents our empirical analysis using the IFC metric. Appendix B gives full details of the IFC calculation.

IFC enables us to measure *variable re-use*. In SSA, variable names are an infinite resource; we instantiate a new variable name at each definition point in the program text. This is not generally the case in standard CFG code. The CFG representation allows variable recycling, since it does not have a strict renaming scheme like SSA. We say that CFG variable *recycling* occurs when several definitions are multiplexed onto the same name. Sometimes this is due to higher level programmer concerns, for example the definitions all relate to the same concept, such as `currentTemperature`. On the other hand, the recycling may be entirely *co-incidental*, for example the programmer (or compiler) re-uses a `tmp` variable as a place-holder for a non-trivial arithmetic expression. However variable re-use can reduce *precision* in data flow analysis, particularly for flow-insensitive data flow analysis. Less variable re-use should make data flow analysis easier, so IFC scores somehow act as a measure of data flow analysis complexity. We want to compare the IFC metrics for programs in SSA and CFG representations.

4 EXPERIMENTAL INFRASTRUCTURE

All intermediate code used in our experimental analysis is generated by the GNU Compiler Collection (GCC) version 4.2.1, running on x86-64 Linux. The SSA form for each procedure is extracted with the `-fdump-tree-ssa` flag. This code is set to be optimized at the `-O3` level, however the SSA dump occurs before most of the optimization takes place. We construct the equivalent CFG code by simply eliding SSA variable subscripts and ϕ-functions.

We create custom perl scripts to analyse the GCC-generated SSA dumps. Our scripts extract the key parts of each SSA dump, namely operators and operands in each basic block, without needing to parse the entire debug dump information. This partial parsing is accomplished using techniques based on *island grammars* [26]. We treat each GCC virtual operand as a variable, in our metrics. We treat each GCC machine operation at an operator, in our metrics. Appendix A gives a full description of the metrics calculations.

The programs to be analysed are the C language programs in the integer section of the SPEC CPU 2000 benchmark suite [14]. Table 2 summarises the

procedural properties for each benchmark. Sizes are measured in terms of number of distinct basic blocks in each procedure. This measure is invariant of the CFG/SSA transformation. Note that the *176.gcc* benchmark is by far the largest in terms of number of procedures to analyse; it also has the largest single procedure.

Table 2. Table of SPEC benchmarks used in experiments, with details of their procedure sizes

benchmark	*# procs*	*proc size*		
		min	*max*	*mean*
164.gzip	67	1	148	27.6
175.vpr	175	0	547	29.9
176.gcc	1807	1	1821	46.1
181.mcf	26	1	106	20.2
186.crafty	39	1	370	60.0
197.parser	323	1	435	27.3
254.gap	698	1	204	29.2
255.vortex	499	1	213	17.5
256.bzip2	74	1	214	17.8
300.twolf	190	1	409	47.3

5 Analysis

This section reports on several metric comparisons between CFG and SSA, using the software metrics described in Section 3 and the tools described in Section 4.

5.1 Halstead Effort Comparisons

This investigation studies the relative difference in Halstead effort between CFG and SSA versions of the each procedure.

The average *ratio* of SSA effort to CFG effort can be computed by dividing the total SSA effort for all 3893 procedures by the total CFG effort for the same procedures. In effect, this is an arithmetic mean, where each procedure's contribution is *weighted* by its original CFG effort. Such a calculation generally rewards reductions for *higher effort* procedures, since these can cause greater efficiency savings. This average SSA:CFG ratio is computed as 0.63, which means that the SSA effort is 37% lower than the CFG effort.

Figure 1 shows how the CFG to SSA Halstead effort ratio changes with program size. Again note the logarithmic scale on the x-axis. Each procedure is denoted by a single (x, y) point, where x is the procedure size and y is the CFG:SSA effort ratio. Points above the line $y = 1$ indicate that the procedure has a lower complexity in SSA than in CFG form. The *majority* of points fall above

$y = 1$. The best fit curve is a line $y = ax + b$, computed by linear least-squares regression. This best fit curve indicates that the magnitude of the complexity reduction for SSA transformation *increases* with procedure size.

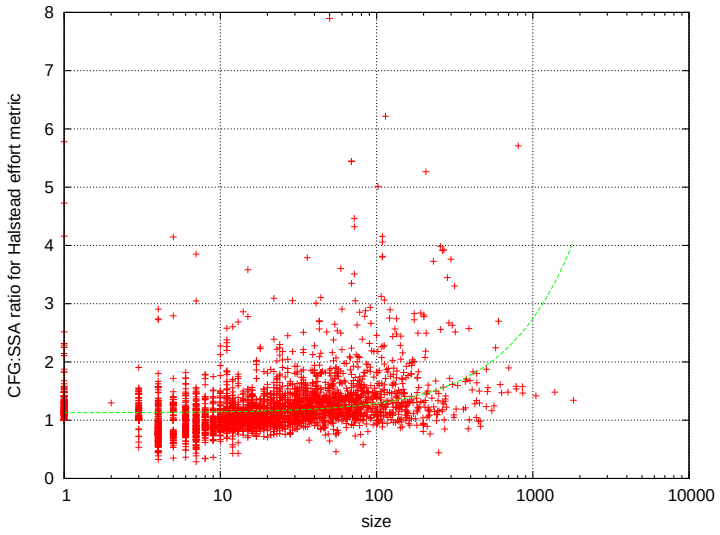

Fig. 1. Correlation of CFG:SSA Halstead effort ratios with procedure size

We report the reduction in total effort for each benchmark program as a result of the SSA transformation. For a single benchmark b, this value is computed by summing the CFG Halstead effort scores for all procedures in b, then summing the SSA Halstead effort scores for all procedures in b, then reporting the relative value of the sum of SSA scores in relation to the sum of CFG scores. Thus a relative value of 1.0 means the SSA transformation does not affect Halstead complexity, for this benchmark. A relative value *below* 1.0 means that the SSA transformation *reduces* the Halstead complexity, for this benchmark. Figure 2 shows these results. The Halstead effort is reduced by the SSA transformation, for all benchmarks. As shown in the graph, the geometric mean of the complexity ratio is 0.68, which means that the average complexity reduction for a SPEC benchmark program in SSA form is 32%, according to Halstead's effort metric.

5.2 Information Flow Complexity Comparison

The final investigation studies the relationship between IFC scores for SSA and CFG versions of each basic block in each procedure.

From our set of benchmarks, there are 3893 procedures. The total number of basic blocks over all these procedures is 141513. We *adjust* the IFC metrics

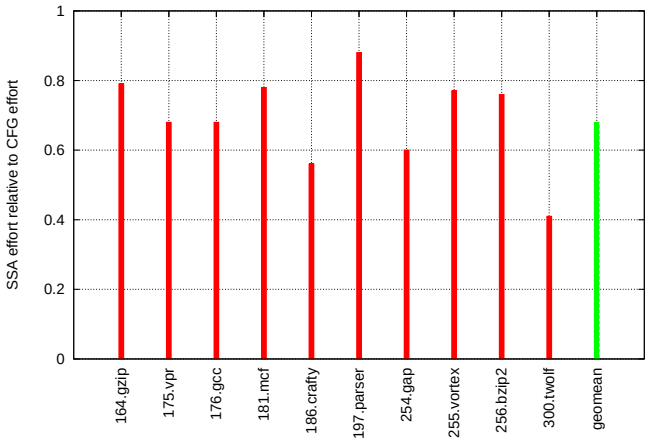

Fig. 2. Reduction in Halstead Effort after the SSA transformation (smaller is better).

scores by adding one to each individual score. This avoids problems with zero values when we compute SSA:CFG ratios and plot log-scale graphs.

The mean SSA:CFG ratio for IFC over all analysed basic blocks is 0.91. Figure 3 shows the results. There is one (x, y) point for each basic block. The x-axis gives the adjusted IFC score for the SSA version of a basic block; the y-axis gives the adjusted IFC score for the corresponding CFG version. Note that 90% of basic blocks have the same IFC score for both IRs. Where there is a difference, then the CFG score is always higher than the SSA score. Thus all points are on or above the line $y = x$. Note the logarithmic scales on both axes in this graph.

We had expected more than 10% of the basic blocks to have lower IFC for SSA than for CFG. We account for this because (i) many basic blocks are extremely small and simple, so there is no difference between the CFG and SSA forms; and (ii) in the GCC compiler, the CFG creation phase is allowed to use unlimited virtual operands, like SSA creation (although without the possibility of ϕ-functions). As Cooper and Torczon explain [8] this is an attempt to reduce incidental sharing. However, there is still some reuse that can only be eliminated by transformation from CFG to SSA.

There are several interesting trends in the graph in Figure 3. For instance, some basic blocks have an IFC score of zero for SSA; although such scores are adjusted to one for reasons mentioned earlier. This anomaly generally occurs because the SSA fanOut score for these basic blocks is zero, implying that there are no outgoing live variables from the blocks in SSA. Such blocks define only global variables. Due to issues of pointer aliasing and inter-procedural optimization, the GCC variant of SSA gives each use of a global variable a separate subscripted virtual operand name. Thus definitions of global variables (marked with the `VDEF`

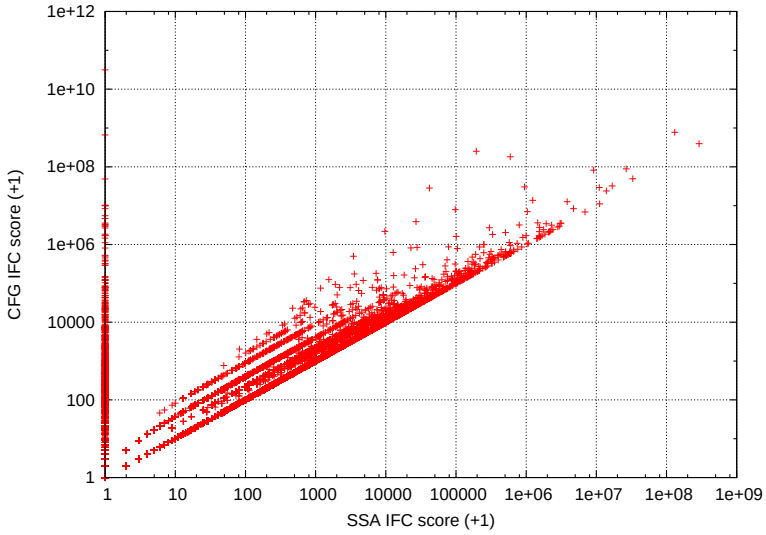

Fig. 3. Comparison of Information Flow Complexity scores for basic blocks in SSA and CFG forms

pseudo-operation in GCC's SSA form) appear to be defining variables that are never used. Thus the fanOut is computed to be zero. It is legitimate to question whether this interpretation of IFC is fair. We argue that the answer is *yes*, since separate subscripts enable data flow disambiguation, which makes it easier to analyse inter-procedural data flows for global variables. In contrast, the CFG representation does not split namespaces for global variables, so inter-procedural analysis is correspondingly less efficient or effective.

For the 10% of basic blocks that have greater IFC for CFG than SSA, in some cases the CFG IFC score is *orders of magnitude* greater, due to the squared term in the IFC equation.

As with Halstead effort above, we also report the reduction in metrics scores for each individual benchmark program as a result of the SSA transformation. For a single benchmark b, this value is computed by summing the CFG IFC scores for all basic blocks in b, then summing the SSA IFC scores for all basic blocks in b, then reporting the relative value of the sum of SSA scores in relation to the sum of CFG scores. Thus a relative value of 1.0 means the SSA transformation does not affect IFC. A relative value *below* 1.0 means that the SSA transformation reduces IFC. Figure 4 shows these results. The IFC metric is reduced by the SSA transformation, for all benchmarks. The geometric mean of the SSA:CFG complexity ratio is 0.4, which means that the average complexity reduction for a SPEC benchmark program in SSA form is 60%, according to the IFC metric.

Note that the complexity reduction is most dramatic for *176.gcc* and *254.gap*. These benchmarks have several data structure initialization procedures with

extremely long basic blocks. The long blocks have high CFG fanOut scores but significantly lower SSA fanOut scores. The squared term in the IFC equation accounts for the massive difference in total complexity.

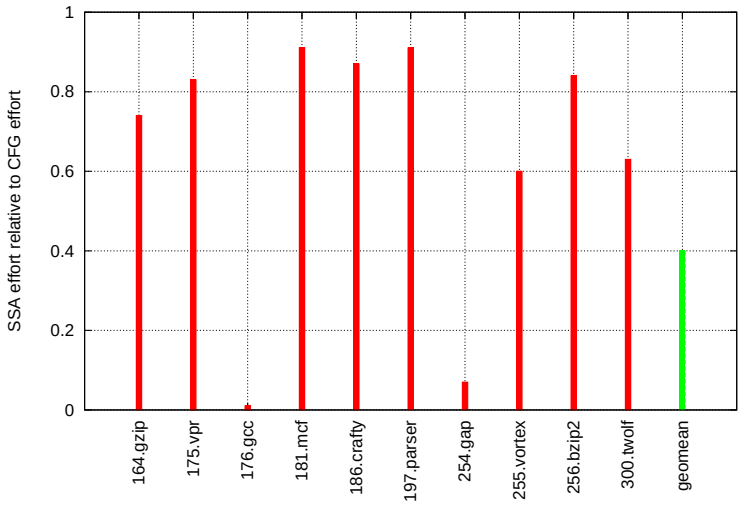

Fig. 4. Reduction in Information Flow Complexity after the SSA transformation (smaller is better).

6 Discussion

The analyses in Section 5 show that the translation of a program from CFG to SSA generally *reduces its complexity* according to two standard metrics.

1. The average per-benchmark Halstead effort metric is 32% lower.
2. The average per-benchmark IFC metric is 60% lower.

Low scores for Halstead and IFC metrics usually indicate well-constructed and easily maintained code. This observation is conventionally applied to high-level source code, where low metrics values indicate that the code should be straightforward for programmers to *understand* and *modify*. We argue that the same observation may be applied to low-level intermediate code, where low metrics values indicate that the code should be straightforward for compilers to *analyse* and *optimize*. Therefore, given the results of our metrics-based analyses in Section 5, we conclude that SSA code is easier for compilers to optimize than equivalent CFG code. Intuitively, SSA-based optimization is simpler and more elegant.

The experience of the compiler construction community harmonizes with such a conclusion. They also observe that SSA facilitates better compiler analysis and optimization passes. For instance, Muchnick [27] states:

> SSA ... simplifies and makes more effective several kinds of optimizing transformations, including constant propagation, value numbering, invariant code motion and removal, strength reduction, and partial redundancy elimination.

Again, compiler researchers prefer SSA over CFG because SSA introduces more local variables, which can lead to more precise data flow information being associated with each variable. This is especially true for *flow-insensitive analysis*, where one unit of data flow information is stored for each variable globally; which contrasts with *flow-sensitive* analysis, where there are N_v units of data flow information for each variable v (one for each of the N_v locations where v is in scope). For example, Hasti and Horwitz [13] show that the SSA transformation makes flow-insensitive pointer analysis as accurate as flow-sensitive analysis in some cases. Our metrics-based analysis reinforces this intuition: that SSA is superior to CFG because it has more variables. Halstead's effort metric captures this property, since the effort score is inversely proportional to the number of unique variable names (n_2). The original motivation is that information is assumed to be spread uniformly over all the variable names, which makes it easier analysis to generate precise information about a single name when there are *more variable names*.

Some members of the compiler community raise the objection that SSA adds overhead to the IR, not only due to the expanded vocabulary but also because of the many ϕ-function insertions required. To quote Muchnick [27] again:

> The occasional large increase in program size is of some concern in using SSA form but, given that it makes some optimizations much more effective, it is usually worthwhile.

This is clearly a valid concern; there is some baggage that comes with SSA. However as Muchnick states, the benefit outweighs the cost in general. This tradeoff is obvious in the Halstead metrics. The *volume* metric increases with the CFG to SSA transformation, due to the extra infrastructure. However the *difficulty* metric decreases due to the increased vocabulary. As observed in Section 3.2, difficulty decreases at a faster rate that volume increases; so the *effort* (which is the product of volume and difficulty) also decreases in general.

Another common qualitative justification for SSA is that it localises data flows, due to aggressive *live range splitting* [5, 32]. This reduces accidental sharing of variable names. It means that variable optimizations may have fewer global side-effects. In SSA, data flow optimizations can often occur at the peephole optimization level due to the localization of data flows. Our new interpretation of the IFC metric (at the basic block level) quantifies this localization. Our analysis shows that the CFG to SSA transform does reduce static data flow dependences between basic blocks.

We note one *caveat* from our analyses. As mentioned earlier, the GCC translator from high-level source code to CFG intermediate code (known as the *gimplifier* [25, 30]) is allowed to introduce unlimited new variable names. Although the CFG IR does *not* have the single assignment property before its conversion to SSA, it often has simplified live ranges in relation to the original high-level program. Thus our metrics-based complexity reduction from CFG to SSA should be treated as a *lower bound*. If the gimplifier did not introduce large numbers of new variable names, then the complexity reduction would be even greater for SSA.

However perhaps this would not be a fair evaluation of CFG, since its variable naming convention is often far from naive. For instance, Cooper and Torczon [8] motivate and describe their CFG-based variable naming scheme for a 1980's FORTRAN compiler before they became aware of SSA:

> Unfortunately, associating multiple expressions with a single temporary name obscured the flow of data and degraded the quality of the optimization. The decline in code quality overshadowed any compile-time benefits. Further experimentation led to a short set of [variable renaming] rules that yielded strong optimization while mitigating growth in the name space. ... The compiler used these rules until we adopted SSA form, which has its own naming discipline.

In a similar way, the CFG intermediate code generation process in GCC does some principled variable renaming. Nevertheless, no matter how much variable renaming is applied to CFG; apart from the SSA constraint, there is still potential for unnecessary inefficiency in compiler analyses.

So, if it were needed, this paper provides clear quantitative evidence to endorse the adoption of SSA in GCC.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, 2nd edn. (2006)
2. Amme, W., von Ronne, J., Franz, M.: Quantifying the benefits of ssa-based mobile code. Electronic Notes in Theoretical Computer Science 141(2), 103–119 (2005)
3. Appel, A.W.: SSA is functional programming. ACM SIGPLAN Notices 33(4), 17–20 (Apr 1998)
4. Braun, M., Hack, S.: Register spilling and live-range splitting for SSA-form programs. In: Proceedings of the International Conference on Compiler Construction. Lecture Notes in Computer Science, vol. 5501. Springer (2009)
5. Briggs, P.: Register allocation via graph coloring. Ph.D. thesis, Rice University (1992)
6. Burke, M.G., Choi, J.D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M.J., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The jalapeño dynamic optimizing compiler for java. In: Proceedings of the ACM 1999 Conference on Java Grande. pp. 129–141 (1999)
7. Chidamber, S., Kemerer, C.: A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20(6), 476–493 (1994)

8. Cooper, K.D., Torczon, L.: Engineering a Compiler. Morgan Kaufmann (2004)
9. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method of computing static single assignment form. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 25–35 (1989)
10. 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 Transactions on Programming Languages and Systems 13(4), 451–490 (Oct 1991)
11. Fitzgerald, R., Knoblock, T., Ruf, E., Steensgaard, B., Tarditi, D.: Marmot: An optimizing compiler for Java. Software: Practice and Experience 30(3), 199–232 (2000)
12. Halstead, M.H.: Elements of Software Science. Elsevier (1977)
13. Hasti, R., Horwitz, S.: Using static single assignment form to improve flow-insensitive pointer analysis. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 97–105 (1998)
14. Henning, J.L.: SPEC CPU2000: Measuring CPU performance in the new millennium. IEEE Computer 33(7), 28–35 (2000)
15. Henry, S., Kafura, K.: Software structure metrics based on information flow. IEEE Transactions on Software Engineering 7(5), 510–518 (1981)
16. Henry, S., Selig, C.: Predicting source-code complexity at the design stage. IEEE software 7(2), 36–44 (1990)
17. Holloway, G.: The Machine-SUIF static single assignment library (2001), `http://www.eecs.harvard.edu/hube/software/nci/ssa.html`
18. Ince, D., Shepperd, M.: An empirical and theoretical analysis of an information flow-based system design metric. In: Proceedings of the European Conference on Software Engineering. Lecture Notes in Computer Science, vol. 387, pp. 86–99 (1989)
19. Johnson, R., Pingali, K.: Dependence-based program analysis. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 78–89 (1993)
20. Kelsey, R.A.: A correspondence between continuation passing style and static single assignment form. ACM SIGPLAN Notices 30(3), 13–22 (Mar 1995)
21. Laird, L.M., Brennan, M.C.: Software measurement and estimation: a practical approach. Wiley (2006)
22. Liao, S.W., Diwan, A., Bosch, Jr., R.P., Ghuloum, A., Lam, M.S.: SUIF explorer: an interactive and interprocedural parallelizer. In: Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 37–48 (1999)
23. McCabe, T.: A complexity measure. Proceedings of the 2nd International Conference on Software Engineering (1976)
24. McDonald, P., Strickland, D., Wildman, C.: Estimating the effective size of auto-generated code in a large software project. In: Proceedings of the 17th International Forum on COCOMO and Software Cost Modeling (2002)
25. Merrill, J.: GENERIC and GIMPLE: A new tree representation for entire functions. In: Proceedings of the First Annual GCC Developers' Summit. pp. 171–179 (2003), `ftp://gcc.gnu.org/pub/gcc/summit/2003/GENERIC%20and%20GIMPLE.pdf`
26. Moonen, L.: Generating robust parsers using island grammars. In: Proceedings of the Eighth Working Conference on Reverse Engineering. pp. 13–22 (2001)
27. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)

28. Mycroft, A.: Type-based decompilation. In: Proceedings of the 8th European Symposium on Programming. Lecture Notes in Computer Science, vol. 1576, pp. 208–223. Springer (1999)
29. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999)
30. Novillo, D.: Tree SSA—a new optimization infrastructure for GCC. In: Proceedings of the First Annual GCC Developers' Summit. pp. 181–193 (2003), http://www.airs.com/dnovillo/Papers/tree-ssa-gccs03.pdf
31. Novillo, D.: Design and implementation of Tree SSA. In: Proceedings of the Second Annual GCC Developers' Summit. pp. 119–130 (2004), http://www.airs.com/dnovillo/Papers/tree-ssa-gcc2004.pdf
32. Quintão Pereira, F.M., Palsberg, J.: Register allocation by puzzle solving. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 216–226 (2008)
33. Schouten, D., Tian, X., Bik, A., Girkar, M.: Inside the Intel compiler. Linux Journal 2003(106), 4 (2003), http://www.linuxjournal.com/article/4885
34. Staiger, S., Vogel, G., Keul, S., Wiebe, E.: Interprocedural Static Single Assignment Form. In: Proceedings of the 14th Working Conference on Reverse Engineering. pp. 1–10 (2007)
35. Succi, G., Liu, E.: A relations-based approach for simplifying metrics extraction. Applied Computing Review 7(3), 27–32 (1999)
36. Ward, M., Bennett, K.: Formal methods to aid the evolution of software. International Journal of Software Engineering and Knowledge Engineering 5(1), 25–47 (1995)
37. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Transactions on Programming Languages and Systems 13(2), 181–210 (Apr 1991)

A Operators and Operands in our Metrics Calculations

We have precise definitions of what constitutes an operator or an operand, for the GCC intermediate code that we analyse. This allows us to apply the Halstead metrics consistently to all procedures that we analyse.

Operands are real and virtual statement operands from GCC Tree SSA [31]. A *real* operand represents a single, non-aliased, memory location which is atomically read or modified by a statement (i.e., variables of non-aggregate types whose address is not taken). A *virtual* operand represents either a partial or aliased reference (i.e., structures, unions, pointer dereferences and aliased variables).

Operators are basic GIMPLE three-address operators, as outlined in Table 3. Each operator takes in one or more operand values and performs a single computational operation.

B Calculation of Information Flow Complexity Metric

The IFC metric value for a function (that was originally a basic block in an SSA procedure) depends on three fundamental measures: length, fanIn and fanOut.

Table 3. GIMPLE operators

operator(s)	*description*
[ ]	array subscript
+, −, ∗, /	arithmetic, pointer deref
&, |, ˜, ˆ	logical
<<, >>	shift
==, ! =, <, >, <=, >=	comparison
`case`	switch statement test
`VUSE, VMAYDEF, VMUSTDEF,PHI`	Tree SSA pseudo-operators

The length is computed as the number of GIMPLE operators in the function. Since GIMPLE is a three-address code, there is generally one operator per instruction.

Given a function f that is derived from original basic block b, the fanIn is computed as the number of input parameters for f. There is one input parameter to f for each variable that has an upwardly exposed use in b, whose variable definition dominates b. Again, there is one input parameter to f for each variable that is defined by a ϕ-function at the head of b. (Appel [3] explains how variables defined on the left hand side of ϕ-functions become formal parameters in function definitions, and variables used on the right hand side of ϕ-functions become actual parameters in function calls.)

The fanOut for f is computed as the number of variables defined in f that are used in another function, plus the number of return values for f. Note that some functions define variables that are never used elsewhere. This may be because the live range of such a variable is restricted that single function. Another reason is that GCC has special *virtual operands*. Each aliased memory location has a distinctly named virtual operand for every static occurrence of that location: i.e. each aliased location has an associated virtual operand that is redefined at every potential definition or use of that location. Such virtual operands may appear to be defined and never used. Section 5.2 discusses how we handle such operands.