One Year of Experience with SUIF

Andrea Olgiati, Stuart V. Quick, John W. Lumley
Appliance Computing Department
HP Laboratories Bristol
HPL-97-86
July, 1997

compiler SUIF

This paper reports on some recent use of the SUIF compilation suite at Hewlett-Packard Laboratories, Bristol. Our main goal is to provide feedback and suggestions to the SUIF Compiler Group for further improvements, based on our experience across a number of projects.

Internal Accession Date Only

One Year Of Experience With SUIF

Andrea Olgiati, Stuart V. Quick and John W. Lumley
Hewlett-Packard Laboratories
Filton Road, Stoke Gifford, Bristol BS12 6QZ, United Kingdom
e-mail: {aol|svq|jwl}@hplb.hpl.hp.com

Abstract

This paper reports on some recent use of the SUIF compilation suite at Hewlett-Packard Laboratories, Bristol. Our main goal is to provide feedback and suggestions to the SUIF Compiler Group for further improvements, based on our experience across a number of projects.

1. Introduction

Part of HP Labs, Bristol's efforts are devoted to study technologies to enable the development and production of timely, low-cost media-intensive appliances. Some of the research effort is directed towards the exploration and understanding of the compilation issues related to this topic, and SUIF is one of the frameworks used for this purpose.

1.1 SUIF at HP Labs, Bristol

At HP Labs, Bristol, SUIF is not used to generate object code as much as a tool for program transformation and analysis: that is, the net product of the SUIF tools usually consists in a modified version of the input code, and/or various pieces of information about it. These results, in turn, are used for two purposes:

- exploiting particular features of the execution environment in which this code is executed, and
- driving architectural and structural transformations for new HP products.

Nonetheless, there have been cases in which machine code had to be generated - Harvard University's Machine SUIF [HAR96] was used for this purpose, in conjunction with University of Wisconsin's SimpleScalar MIPS simulator [WIS96].
The rest of the paper is structured as following: Sections 2 and 3 contain the description of two different research projects using SUIF, Section 4 has some general considerations and Section 5 the conclusions.

1.2 Compilation Issues

One often-quoted consideration is the rule of thumb known as *the 90/10 locality rule*: a program executes about 90% of its instructions in 10% of its code [HP96]. This is particularly true for one class of programs, which are characterized by one or more computationally intense kernels, which are repeatedly executed; almost all the time, they assume the form of FOR loop bodies. Given their importance and weight relative to other parts of the computation, a great deal of attention is focused on these loop bodies.
Different research projects, in which SUIF is involved, are carried out at HP Labs, Bristol; they can be classified in two main categories:

- code analysis;
- code modification for architectural exploitation and enhancement.

The first research project described (Dynamic Dataflow Characterization - Section 2) is an example of code analysis, while the second (Packed Arithmetic Exploitation - Section 3) is related to the exploitation of an architectural feature of a particular class of microprocessors.

2. Dynamic Dataflow Characterization (DDC)

DDC is a toolchain whose purpose is the characterization of computational patterns contained in loop bodies. DDC extracts these patterns from selected parts of C source files (in the form of dataflow graphs) and passes them to an external analysis tool. In this context, SUIF was used to:

- generate dataflows from the loop bodies;
- output dataflow descriptions in VCG [VCG96] and proprietary format for non-SUIF processing;
- reinsert the processed dataflows information into the SUIF files for further processing.

The analysis process had to be carried out outside the SUIF framework for eminently practical reasons:

- interaction with the user was needed via a Tcl/Tk graphical interface - integrating these functionalities in the SUIF environment would have further complicated the task;
- graph manipulation facilities are not provided in the core SUIF distribution.

2.1 SUIF Characteristics Discovered in DDC

2.1.1 Interchangeability

The possibility of intermixing SUIF and non-SUIF passes while retaining temporary results inside the saved SUIF files and the ability to pick an almost arbitrary order of application of these passes were vital to DDC: they allowed the use of the best execution environment for each part of the work. Nonetheless, some difficulties emerged, as discussed in the next point.

2.1.2 Difficulties in Retaining Non-Standard Information

Reading and writing arbitrary information from/to a file, within a SUIF pass, is quite a challenge, since a non-trivial mechanism called *structured annotations* has to be used. In this case, furthermore, dataflows represented by pointers had to be turned into indices 'pointing to' specific trees in a procedure before saving the SUIF file. The dynamics of the process of reading in SUIF files do not allow the inverse transformation to happen easily, so this operation had to be completed in a subsequent step.

2.1.3 Lack of Support for Graph Building and Analysis

Many (standard and non) compiler passes rely on graph manipulation techniques to analyze and transform dataflow graphs and solve dataflow equations, and so does DDC. Rather surprisingly, the only feature offered by SUIF to implement or customize these passes is the code contained in a tool called `nsharlit`. Although very powerful, `nsharlit` is extremely difficult to master, and, for small cases, sometimes it is not worth the effort. Some documentation exists [TJI92] [TJI93], but it is rather out-of-sync with the current distribution.

3. Packed Arithmetic Exploitation (PAE)

In this project, SUIF was used as a principal program transformation tool to investigate the use of the SIMD instructions incorporated in modern microprocessors. This involves focussing on FOR loop bodies, examining extractable parallelism, principally by strip-mining or transforming to strip-minable forms, determining data sizes and then systematically replacing data with packed forms and operations with their packed equivalents, or equivalent sequences in the available instruction set. Currently, designer interaction before these phases is used to focus effect and clear 'safety' ambiguities such as those arising from pointers, data sizes or the applicability of specialist techniques (e.g. saturated arithmetic). The final output is then back-transformed to C-source using in-line assembly directives for the invocation of the SIMD instructions.
SUIF was first employed for its ability to preserve the loop structures on which the techniques focused. For this experimental work complete compilation was not required, but systematic transformation of loop headers and bodies could be implemented with some fairly small programs exploiting the various SUIF classes.
The size of the codes transformed was not large and the performance of the various SUIF passes wasn't a problem, though scaling issues weren't really addressed. What was important was the speed of development.

This work used SUIF and its libraries as a small-scale, user guided, program transformation tool. As such it allowed us to focus on the core of the transformations to explore, hiding the details involved with building a full-blown compiler. At the small scale and limited scope involved it proved adequate, as it allowed us to demonstrate successful and correct transformations.

3.1 SUIF Characteristics Discovered in PAE

3.1.1 Extensibility

Using `in_gen` meant that new operators corresponding to the SIMD instructions could be created readily, and with annotations for `s2c` result in 'automatic' generation of appropriate C source. This turned out to be extremely useful.

3.1.2 Type Manipulation

Formation of new union types corresponding to 'packed' variables could be built quickly and appropriate systematic substitutions within the variable and constant space made quite simply.

3.1.3 Annotations

The ability to place arbitrary information on annotations to record intermediate states (such as variable substitution mappings) and place comments, `#pragmas` and so forth in the final output proved very useful. Various access functions needed to be built however to simplify the 'simple' use of annotations.

3.1.4 Tree Matching

One function that might perhaps have turned out extremely useful was a package supporting 'tree pattern' matching and substitution; not to the extent of unification, but perhaps to the level of regular expression substitution. Then a lot of the 'tree-walking' codes could have been simple (possibly interpretative) library calls.

4. General Points

Amongst the more common problems and opportunities for improvement, the ones that we think worth mentioning are:

4.1.1 Redundancy of the Two-Layered Structure

SUIF's internal representation is based upon two layers of objects: trees and instructions. Writing functions that perform the traversal of both these structures means duplicating a lot of code, thus increasing the development and maintenance effort.

4.1.2 Incompatibility with Existing Development Tools

Especially in the case of large and articulate projects, sometimes only some of the source files need to be processed by a SUIF pass, whereas other files are dealt with using more conventional development toolchains. Some problems arose while integrating the results of a pass built using SUIF, whose results were MIPS assembly files, with the output of `gcc`, as different alignment paradigms were used.

4.1.3 Instruction Formats

We think that the instruction format used in SUIF could be improved giving it a more consistent appearance: for instance, some instructions have destination operands whereas others do not. A possibility might be to replace the destination operand with `ld`-like instruction. This would generate a larger number of instructions in a program but with fewer inconsistencies. A construct like

```
add foo = e1, e2
```

would, then, become

```
ld foo, e0
e0: add e1, e2
```

4.1.4 Interaction with `builder`

This was less smooth than hoped. What appeared on the surface to be cases where `builder` could be used to construct a fairly large 'expression' and then splice to appropriate parts of the tree, turned out to be a lot more problematic and in the end simpler to construct 'by hand' in SUIF. Furthermore, at the moment, `builder` cannot create code such as `foo[i]->bar`. This could be a real problem while generating code that communicates with hardware or special register sets.

4.1.5 `#include` Files and `s2c`

During the parsing process, information regarding which files are `#included` is lost - this results in the generation of large `.c` files from within `s2c`. Retaining the original `#include` directives instead would generate smaller `.c` files, and would be very useful in subsequent preprocessing phases.

5. Conclusions

In general, our work experience with SUIF has proven positive. The annotations we list in this paper are suggestions for further improvements to the SUIF compilation suite.

6. References

[HAR96] *Machine SUIF*, HUBE Research Group, Division of Engineering and Computer Science, Harvard University, 1996 - `http://www.eecs.harvard.edu/~hube/#software`

[HP96] *Computer Architecture: A Quantitative Approach*, 2nd Edition, John L. Hennessy, David A. Patterson, Morgan Kaufmann Publishers, San Francisco, CA, USA, 1996

[TJI92] *Sharlit - A Tool for Building Optimizers*, Steven W. K. Tjiang, John L. Hennessy, Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, July, 1992 - `http://suif.stanford.edu/papers/tjiang92.ps`

[TJI93] *Automatic Generation of Data-flow Analyzers: A Tool for Building Optimizers*, Steven W. K. Tjiang, Ph.D. Thesis, Stanford University, Computer Systems Laboratory, July, 1993 - `http://suif.stanford.edu/papers/tjiang93.ps`

[VCG96] *Visualization of Compiler Graphs*, Georg Sander, Universität des Saarlandes, Saarbrücken, Germany, 1996 - `http://www.cs.uni-sb.de/RW/users/sander/html/gsvcg1.html`

[WIS96] *The Multiscalar Project*, Computer Sciences Department, University of Wisconsin, 1996 - `http://www.cs.wisc.edu/~mscalar/`