Non-Transparent Debugging for Software-Pipelined Loops

Hugo Venturini, Frédéric Riss, Jean-Claude Fernandez, Miguel Santana

Non-Transparent Debugging For Software-Pipelined Loops

Hugo Venturini*
hugo.venturini@imag.fr
Verimag / STMicroelectronics

Frédéric Riss
frederic.riss@st.com
STMicroelectronics

Jean-Claude Fernandez
jean-claude.fernandez@imag.fr
Verimag

Miguel Santana
miguel.santana@st.com
STMicroelectronics

ABSTRACT

This paper tackles the problem of providing correct information about program variable values in a software-pipelined loop through a non-transparent debugging approach. Since modern processors provide instruction level parallelism, software pipelining techniques have been developed to achieve better performances, especially in the context of embedded systems. Indeed, the effectiveness of software pipelining on such systems has been demonstrated both theoretically and experimentally. As it overlaps iterations and reorders statements, it also makes standard debugging information irrelevant. Hence debugging a loop which has been software-pipelined becomes very difficult. In this paper, we propose a solution relying on selected information to be generated by the compiler and an algorithm for the debugger not to mislead the user.

ACM Classification

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation

1. KEYWORDS

compiler, debugger, non-transparent debugging, software-pipelining

2. INTRODUCTION

Over the last thirty years, program optimizers' technologies improved significantly faster than debuggers' technologies. This disparity is reflected in the literature; many books and articles treat optimizations but very few address the debugging of optimized code. Nevertheless, in the context of embedded systems, debugging the optimized code is a necessity. The programmer may need to debug his program in real-life conditions, i.e. onto the embedded systems which may not accept the unoptimized program due to constraints. Optimizations are often required because of the real time and memory constraints imposed, especially on embedded systems. Moreover, the overhaul will be done on an optimized program so the programmer has to test and debug the latter on the device it is targeted to.

A wide range of processors, e.g. superscalars or VLIW (Very Long Instruction Word) use parallelism mechanisms which can be hardware and software. One of these optimizing methods is called *software-pipelining*. It is performed by the compiler and allows the use of parallelism inherent in an application.

Software pipelining is a type of instruction scheduling where the goal is to construct an equivalent loop of minimum length by overlapping computations from different iterations of the original loop. Hence, it makes standard debug information obsolete. Debugging a loop which has been software-pipelined would not make sense without a correct mapping between the source code and the assembly code. Instructions generated from various source statements are duplicated, combined, moved, deleted and interleaved with instructions from other source statements. It becomes very difficult to decide where in the target program any given source statement begins or ends. Thus, breakpoints set in the source code may not find their equivalent in the optimized program and vice versa. Debugging the optimized code becomes very complicated.

We propose an approach in which the compiler generates more information than it does under the current standard. We have designed an algorithm for the debugger to accurately use this information and answer the following question: *When retrieving the value of a variable at a given address, which iteration count produced the value?* The idea is not to hide anything from the user, but rather to guide the user through the debugging process. When a variable's assignment has been delayed, not only do we provide the variable's current value, but we also provide the delay of assignment. Hence the user knows exactly the behavior of the software-pipelined program. The experimentation context is a retargetable compiler for embedded processors which aims at providing state-of-the-art optimizations: MMDSP+ C Compiler. It is built around the CoSy compiler development suite [2] with an in-house back-end [5].

The paper is organized as follows: Section 3 describes software-pipelining loops and provides useful definitions and an example to

*supported by a CIFRE STMicroelectronics industrial grant

CASES'07, September 30–October 3, 2007, Salzburg, Austria.

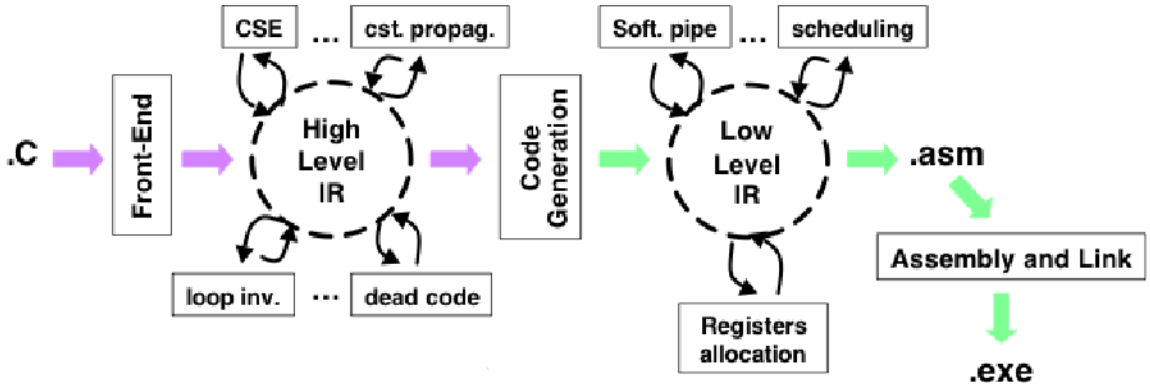

Figure 1: MMDSP+ C Compiler Architecture

illustrate it. Section 4 reviews existing problems due to optimizing programs. Section 5 presents the contributions of our work. Section 6 details debug information to be collected by the compiler. Section 7 describes the algorithm to be implemented by the debugger to solve the issue. We give the tool framework and experimentation in section 8. Section 9 reviews related works since 1982. Finally section 10 concludes and discusses future works.

3. SOFTWARE PIPELINING

3.1 Overview

Software pipelining is a type of instruction scheduling where the goal is to construct an equivalent loop of minimum length by overlapping computations from different iterations of the original loop. It can be used on many architectures, especially those which allow instruction-level parallelism (ILP), but it is particularly efficient on VLIW processors. One VLIW instruction encodes multiple operations; specifically, one instruction encodes at least one operation for each execution unit of the device. For example, if a VLIW device has five execution units, then a VLIW instruction for that device would have five operation fields, each field specifying what operation should be done on that corresponding execution unit.

The software-pipelining algorithm we rely on is the one implemented by the MMDSP+ C Compiler [5]. It is a modulo software-pipelining algorithm inspired by B.Rau [15]. An engine implements an optimization; it selects loops to be software-pipelined. The selection is based on the following criteria:

DEFINITION 3.1. *A loop is eligible if it respects the three following properties:*

- *it does not nest any loop,*
- *its index runs by* 1*s,*
- *it does not contain any* `if`*-statements.*

The last restriction states that the loop body should not contain `if`-statements other than exit tests. Generally, this is a major restriction; developers need to use `if`-statement within a loop body. In the context of embedded systems, applications are specific: the MMDSP processor is dedicated to the encoding and decoding of sound and video formats e.g. amr, mp3. This kind of algorithm makes intensive use of loops which process huge arrays without any `if`-statements. Note that in a more general case, if-conversion, which can be performed on many predicated VLIWs, greatly lowers the impact of this limitation.

The more instructions can be executed simultaneously, the faster the program will run. The fastest possible execution would be to execute all the instructions of the program in parallel. This is impossible because of the following constraints:

- Data dependency: If instruction `A` calculates a result that is used as an operand of instruction `B`, then `B` cannot execute before `A` is finished.
- Functional unit: If there are x multipliers (adders, etc) on the chip, then at most x multiplication (addition, etc) instructions can execute at once.
- Instruction unit: The instruction-issue unit can issue at most y instructions at a time.
- Registers: At most z registers can be in use at the same time.

The three last constraints are often lumped together as resource constraints. This is why the software-pipelining optimization is performed at the assembly-level in the compiler. The first constraint, data dependency, is the only one which affects our work. A dependency between two instructions exists if interchanging their order changes the results. A **Data Dependency Graph** — DDG — is used to describe dependencies between instructions: nodes are operations and the set of edges is the set of dependencies. Anti and output dependencies also are respected during scheduling of the loop body.

Overlapping instructions implies in many cases the generation of a prologue and an epilogue to the loop body. Because instructions are overlapped through iterations of the original loop, some instances of instructions may have to be computed beforehand, in the prologue. The same way, the last instances of some instructions may not be computed in the loop body, thus they are moved later, in the epilogue.

3.2 A Practical Example

Freely inspired from A.Appel [4], a source loop is given in figure 2. Originally, it contained array accesses which overcomplicated the reading of the example and do not affect the output.. We removed them because they did not change anything to the modulo software pipelining algorithm while they loaded down the notation.

It is a `for`-loop for which the number of iteration N is unknown at compilation time.

This example is quite complete because it uses all four kinds of data dependency through iterations which can affect the scheduling of instructions. We sum them up in table 1, where indexes represent

		Iterations					
	Addr	1	2	3 (N-2)	4 (N-1)	5 (N)	source line numbers
	0x01	acfj					(2,4,7,10)
	0x02	bd	fj				(3,5,7,10)
	0x03	egh	a				(6,8,9,2)
Prologue	0x04		bc	fj			(3,4,7,10)
	0x05		dg	a			(5,8,2)
	0x06		eh	b	fj		(6,9,3,7,10)
	0x07			cg	a		(4,8,2)
			for i=3 **to** N-2 **{**				(1)
	0x09			d	b		(5,3)
Loop Body	0x0A			eh	g	fj	(6,9,8,7,10)
	0x0B				c	a	(4,2)
			}				(11)
	0x0D				d	b	(5,3)
	0x0E				eh	g	(6,9,8)
Epilogue	0x0F					c	(4)
	0x10					d	(5)
	0x11					eh	(6,9)

Figure 3: Software-pipelined loop schedule

```
1   for (i = 1; i ≤ N; i++) {
2        a = j + b
3        b = a + f
4        c = e + j
5        d = f + c
6        e = b + d
7        f = 42
8        g = b
9        h = d
10       j = 43
11  }
```

Figure 2: A `for`-loop to be software-pipelined

iteration numbers. Some variables such as `f` or `j` do not have any dependencies on any other variable assigned to in the loop. The second case of dependencies is given with variables `e`, `g` and `h` which depend on instances from the same iterations as theirs. Variables `a` and `c` depend on instances from the previous iteration only. The last case of dependencies is given with variables `b` and `d`, they both depend on instances from different iterations. These last two cases enclose the case where an instance depends on the i^{th} and j^{th} previous iterations. Our example assumes instruction words of five

Variable	Depends on		
a_i	j_{i-1}	b_{i-1}	
b_i	a_i	f_{i-1}	
c_i	e_{i-1}	j_{i-1}	
d_i	f_{i-1}	c_i	
e_i	b_i	d_i	
f_i	42		
g_i	b_i		
h_i	d_i		
j_i	43		

Table 1: Variable Dependencies

operations. In order not to confuse the reader, we omited load and store instructions because they do not change the principles of our proposal. The result of the software-pipelining is given in figure 3. Again, in order to facilitate the reading, we consider that each instruction is exactly the size of one addressable unit. The targeting to a specific processor remains trivial.

4. SYMBOLIC DEBUGGING OF OPTIMIZED CODE

Debugging optimized code poses two problems [8, 26], the *data value problem* and the *code location problem*.

4.1 Data Value Problems

The first problem is known as the set of *data value problems*. They are the difficulties involved in finding and returning the value of a variable in response to a user's query.

In order to save registers, some optimizations, thank to the data flow graph, will make several variables unaccessible at some point in the program. The *residency problem* occurs when a variable's value is not accessible. In figure 4, let f and g be two functions which have no side effects. Also assume `x` is defined within `f`. The user may request `x`'s value whenever the program is running in `f`. Now assume `x` is not used after the 4^{th} line. In order to save resources (registers, memory), the compiler may get rid of `x` after the 4^{th} line, i.e. `x` is not stored anywhere, it is lost. During a classical debugging session, the user can request `x`'s value at the end of `f`, but in this case of optimization, the debugger cannot give the value back to the user.

```
1   int f() {
2     int x;
3     // ...
4     b = g(x);
5     // x is not used
6     return b;
7   }
```

Figure 4: Residency Problem Illustration

Another subproblem, called the *data location problem*, occurs when a variable is not located in the expected register or at the expected address. In figure 5, scalarization makes local a global variable by using a temporary variable, here `tmp`. Then, in the loop, `a` exists somewhere up-to-date, but the debugger will not know its value is located in `tmp`

The last subproblem is called the *currency problem*, see figure 6. The variable `tmp` is not used in the loop, instead, it is used once at line 7, for the same assignment. The compiler may decide to optimize it by using a code sinking or a code hoisting optimization,

```
1   int f() {
2     int x;
3     int i;
4     // ...
5
6     a = a + x;
7     for (i=0; i<10; ++i) {
8       a = a + i;
9     }
10
11    // ...
12    return 0;
13  }
```

(a) Before Scalarization

```
1   int f() {
2     int x;
3     int i;
4     // ...
5     tmp = a;
6     tmp = tmp + x;
7     for (i=0; i<10; ++i) {
8       tmp = tmp + i;
9     }
10    a = tmp;
11    // ...
12    return 0;
13  }
```

(b) After Scalarization

Figure 5: Data Location Problem Illustration

i.e. moving the assignment line 7 before or after the loop. During the debugging session, the user requests `tmp`'s value while running in the loop, but since it has been moved, the value will not be the one expected by the user. The currency problem arises when a variable's value might not be the same at the same point in the source program.

```
1   int f() {
2     int i;
3     int tmp;
4     // ...
5     for (i=0; i<10; ++i) {
6       //..
7       tmp = CONST_X;
8     }
9     // ...
10    return 0;
11  }
```

Figure 6: Currency Problem Illustration

4.2 Code Location Problem

The second problem is the *code location problem*. It arises in mapping between locations in the source code and locations in the optimized program.

For instance, dead code elimination generates a one-to-zero relation between the source code and the optimized program, common subexpression elimination generates one-to-many relations, and hardware loops creation generates many-to-one relations.

5. APPROACH AND CONTRIBUTION

We focus hereafter on the problem of *debugging software-pipelined loops*. Our solution answers this simple question: *Given an address and a variable, what is the value of this variable?* The code location problem and the data value problem both arise after software-pipelining a loop. For example, figure 7(a) is a very simple `while`-loop which computes the sum from 0 to `MAX-1` (variable `x`) and the sum of these sums (variable `y`). Assume the target processor has 2-issues instructions. The first idea is to parallelize variable initializations at line 1 and 2. At compile time, the loop is rewritten as in figure 7(b). In order to parallelize instruction executions, lines 5 and 7 of the source loop have been copied before the loop entry, and line 6 of the source loop has been copied after the loop body. Moreover, lines 5 and 6 are executed in parallel. So, `x` is computed once before entering the loop, and then is computed one iteration step forward `y`'s value computation.

```
1   i = 0
2   x = 0
3   y = 0
4   while ( i < MAX )
5     x = x + i
6     y = x + y
7     i = i + 1
```

(a) Source Code

```
0x334   i = 0          x = 0
0x338   y = 0          x = x + i
0x33C   i = i + 1
0x340   while ( i < MAX )
0x344     y = x + y      x = x + i
0x348     i = i + 1
0X34C   y = x + y
```

(b) Rewritten Code

Figure 7: Example of Misled Debugging

Suppose the user is debugging this program using a standard tool such as `gdb` [17]. Suppose also the program stops at address `0x348` in the rewritten code. The debugger gives the hand back to the user stating that it stopped at line 7 in the source code. The user asks for `y` and `i`'s current values. It seems fair for the user to think that `y`'s value corresponds to the current iteration number `i`. This is incorrect. A trace is given in figure 8, `y` equals 4 whereas `y` should be 10. When at address `0x344` in the rewritten code `x` and `y` are computed, `y`'s value is one iteration step backward the current iteration `i`, i.e. `y` is computed for the $(i-1)^{th}$ time. This misleads the user about the execution of the program. We would like the debugger to give the current value of the variable and the iteration it corresponds to.

In this paper, we propose a solution for the user not to be misled in such a situation. Our proposal relies on selected information to be generated by the compiler and an algorithm for the debugger not to mislead the user.

Instance and *offset* are two words we need to define in order to remain clear in the rest of this paper.

DEFINITION 5.1. *An execution of instruction S is called an **instance** of S.*

We refer to an instance by specifying its occurence *number*. The *instance number* of an operation refers to the source iteration num-

```
(gdb) break 7
 Breakpoint 1 at 0x33C and 0x348: file loop.c, line 7.
(gdb) run
 Starting program
 Reading symbols for shared libraries . done

 Breakpoint 1 at loop.c:7
 7                       i = i + 1;
(gdb) continue 3
 Will ignore next 2 crossings of breakpoint 1.
 Continuing.

 Breakpoint 1 at loop.c:7
 7                       i = i + 1;
(gdb) print x
 x = 6
(gdb) print y
 y = 4
```

Figure 8: Example of requesting variables' values when software-pipelined

ber it would have been computed at if the loop were not software-pipelined. This is accurate because each source instruction is considered unique, thus it appears once and only once per source iteration.

DEFINITION 5.2. *The difference between an instance number of an instruction and the iteration number where it is executed is called the **offset** of the corresponding operation.*

Our approach is called non-transparent as originally defined by T. Zellweger [25]. She gave original definitions for transparent behavior, and correct, or non-transparent behavior: a debugger is said to have a transparent behaviour when its responses to user requests concerning the execution of an optimized program are the same as responses would be for an unoptimized version of the program. On the opposite, a debugger is non-transparent, or correct, when it can display, in source program terms, the relevant changes caused by optimization at execution point.

We aim at providing the correct behavior of a program, not the expected behavior. Indeed, we truly believe that a developer of applications targeted to embedded systems must know the real behavior of the program; being aware of the targeted architecture allows the creation of more efficient applications.

If a variable is assigned earlier or later than expected during the execution, we do not try to compute the expected value, but we provide the iteration number for the user to fully understand what iteration in the source code corresponds to the value he requested. The main idea is to statically compute an offset between the current iteration of a loop and the current instance of an instruction belonging to this loop body. This can be done if the offset of computation of an instruction is not dynamically modified during execution. Moreover, it implies that software-pipelined loops respect the following property:

PROPERTY 5.3. *The order of instances of an instruction remains the same in the source code and in the software-pipelined code.*

Our work relies on this ordering property. The modulo software-pipelining algorithm implemented by MMDSP+ C Compiler follows this assumption. Indeed, very informally, the algorithm can be seen as the scheduling of the source loop entirely unrolled w.r.t. true, anti and output dependencies.

One of our assumptions is that debugging information is kept relevant by other optimizations. This may not seem realistic at first. Compiling is applying a sequence of optimizations to a program. It can be seen as a composition of functions. Our work tends to focus on one of these functions, this does not mean that any work does not have to be done on other functions i.e. optimizations. A weakness is that property 5.3 can be invalidated by other optimizations (Common Subexpression Elimination Across Iterations —CSEAI— for instance). That is the reason why we try this approach alone first and then as part of a more important work on information propagation through compilation partly based on Adl-Tabatabai [3] and Tice [20].

Even though much work has been done in the field of software pipelining, to our knowledge this is the first attempt to debug software-pipelined loops in a non-transparent manner.

6. DEBUG INFORMATION

As written by Gough, Ledermann and Elms [11], computer programs often need to be examined to determine the cause of apparent errors, or to gain a better understanding of their structure. This examination is called *debugging*, since its usual objective is the location and removal of program errors (bugs). Compilers such as `gcc` [16] provide debug information for debuggers via standard debugging information formats e.g. DWARF 2 [1]. Debugging information standards are not usually designed explicitly for debugging optimized code. Compilers do not use every feature which would ameliorate the state of debugging of optimized code in industrial tools (see C.Tice [20]). Without considering any specific debugging information standard, we present in this section information our algorithm needs in order to solve the issue we address.

When a compiler optimizes a piece of code, it knows much more than what it provides as debugging information. A trivial example is the dead code elimination case. When the compiler decides to delete an instruction from the program, it does not record it into debugging information, thus the debugger cannot inform the user when he tries to set a breakpoint at this deleted instruction. When the compiler software-pipelines a loop, it computes the DDG, and so knows where each variable is (or will be) located, computes how many iterations are to be overlapped, as well as other information related to the mapping between the source loop and the software-pipelined loop. During software-pipelining optimization, all such information is mandatory to rewrite the loop into three parts, the prologue, the loop body, and the epilogue. Once the information is known, it becomes straightforward to store it instead of discarding it at the end of the optimization. We instrument the compiler to store some of this information.

For each part of the software-pipelined loop (e.g. figure 3), we instrument the compiler to store four values which informally correspond to boundaries of the loop. We store the lowest and the highest iteration numbers for which instructions are executed in that part and we call them **first** and **last iteration number**. Because iterations are overlapped, the highest instance number of instruction of the `prologue` will very often be higher than the first instance number of instructions of the `loop body`. This will happen between the `loop body` and the `epilogue` as well.

Since each part is made of one block of continuous addresses, we store the first and the last addresses which we call **starting** and **ending addresses**. An example of such information is given table 2. The `prologue`'s first assignment belongs to the first iteration while its last assignment, in terms of instance number, belongs to the fourth iteration. We also store its boundaries, i.e. addresses, here from `0x01` to `0x7`. The `epilogue`'s range of iteration depends on the loop total number of iterations and is here from `N-1` to `N` while its range of address goes from `0x0D` to `0x11` Then, we store an integer per instruction: its offset to the current iteration

	firstIter	lastIter	startAddr	endAddr
prologue	1	4	0x01	0x07
loopbody	3	N	0x09	0x0B
epilogue	$(N-1)$	N	0x0D	0x11

Table 2: General Loop Information

Source Line Number	2	3	4	5	6	7	8	9	10
Variable Assigned To	a	b	c	d	e	f	g	h	j
Offset	2	1	1	0	0	2	1	0	2

Table 3: Instruction Information

number in the loop body. In our example, see table 3, the loop body iterates from 3 to $N-2$. Because at the 3^{rd} iteration, we compute the 5^{th} instance of operation assigning to a at line 2, its offset is 2.

There exists several reasons why a program stops during a debugging session e.g. user breakpoint, program aborting, signal trap. Even if they have different mechanisms, they eventually all stop the program at an address. In the remainder of this paper, we shall call this particular address, the *breaking address*. At that point the user can request variables' values: this will be our algorithm's starting point.

7. ALGORITHM

Besides the traditional debug information and the extra information defined previously, debuggers need mechanisms to find their way into this information. In this section, we first clarify our notation system and then describe our algorithm.

7.1 Assumptions and Notations

The algorithm presented here aims at determining precisely for a given variable which source iteration its current value corresponds to. We define structured types, `t_part` and `t_address`, which contain information collected during compilation.

The `t_part` type contains information relative to the three parts of the software-pipelined loop: the prologue, the loop body, and the epilogue. Variables `prologue`, `loopbody` and `epilogue` are used along the algorithm. Access to their information is given with an arrow e.g. `prologue->firstIter`, see table 4.

The `t_address` type contains information corresponding to addresses of the object code. Each address denotes an instruction, thus each address assigns to a set of variables, we note this set `addr->setVar`. Each operation maps to a source instruction. Because a variable can be assigned only once per instruction at a user-visible level, we use a very simple notation for the access to the line number `lnum` corresponding to a variable `v` of `addr->setVar`: we write `addr->v->lnum` and we use the same logic to write `addr->v->offset`.

Type Name		Variable
t_address	->	setVar
t_address->t_variable	->	lnum
t_address	->	offset
t_part	->	firstIter
	->	lastIter
	->	startAddr
	->	endIter

Table 4: Structures Used

We call `i` the iteration variable, its value in the `prologue` or in the `epilogue` does not have to be defined. In our example figure 3, `i`'s range is the interval $[3..N-2]$. The variable of type `t_variable` for which the user requests the value is called `rv`. The breaking address of type `t_address` is noted `ba`. For instance, if the requested variable is assigned to at the breaking address, we use the following notation: `rv` $\in$ `ba->setVar`. Table 4 sums up the data structures and notations used in our algorithm. Table 5 sums up the input variables for the algorithm. Table 6 shows the result format of the algorithm. We also use a temporary address named `tmpAddr` of type `t_address`.

Variable Name	Type	Comment
ba	t_address	breaking address.
i	integer	The iteration number.
rv	t_variable	requested variable.
prologue	t_part	prologue
loopbody	t_part	loop body
epilogue	t_part	epilogue

Table 5: Input Variables

Variable Name	type	Description
result->addr	t_address	Address which last assigned to rv
result->inst	integer	Instance number of rv

Table 6: Output Variable

One `if`-statement has been omitted at the beginning of every one of our three sub-algorithms: `rv` is supposed to be assigned to at least once in the source loop, thus at least once in the loop body of the software-pipelined loop. This assumption allows us to write exit conditions of `while`-loops on the presence of `rv` in `setVar` without fearing infinite loops. And because every loop of the algorithm does not have any nested loop and exits on this condition, we can assure our algorithm works linearly i.e. $\mathrm{O(n)}$ w.r.t. n the number of instruction in the source loop body.

7.2 Pseudo-code

In this section, we give the algorithm in pseudo-code. It helps the debugger to answer the original question: *At a given address and for a given variable, what is the value of this variable?* The breaking address `ba` can be in the prologue, in the loop body or in the epilogue. We present 3 sub-algorithms, one for each case. The entire algorithm in figure 9 could be written more effectively, but we chose to divide it into three parts in order to make it clear to the reader.

```
1  ask_value (t_address ba, t_variable rv) {
2     if (ba ∈ prologue) then
3              // see figure 10
4     else if (ba ∈ loopbody) then
5              // see figure 11
6     else if (ba ∈ epilogue) then
7              // see figure 12
8     else
9              // ba is not set in the loop.
10 }
```

Figure 9: General Algorithm

7.2.1 Prologue

When `ba` is in the prologue, we first look backward for the last address assigned `rv`. We then look for the number of times it has been assigned to by this operation. Checking the line number and not only the variable allows a variable to be assigned more than once in the loop body.

```
1  tmpAddr = ba - 1
2  while (tmpAddr ≥ prologue->startAddr)
3        ∧ (rv ∉ tmpAddr->setVar) do
4       // search backward for rv
5       tmpAddr = tmpAddr-1
6  if (rv ∉ tmpAddr->setVar) then
7       result->addr = null
8       return // rv has not been assigned  yet
9  else
10      result->addr = tmpAddr
11
12 assAddr = result->addr
13
14 result->inst = prologue->firstIter
15 tmpAddr = prologue->startAddr
16 while (tmpAddr ≠ assAddr) do
17     if (rv ∈ tmpAddr->setVar)
18    ∧ (tmpAddr->rv->lnum == assAddr->rv->lnum) then
19             result->inst++
20     tmpAddr = tmpAddr + 1
```

Figure 10: ba is in prologue

```
1  if (i ≤ loopbody->startIter) then
2      tmpAddr = ba - 1
3      while (tmpAddr ≥ prologue->startAddr)
4           ∧ (rv ∉ tmpAddr->setVar) do
5        // search backward for rv
6        tmpAddr = tmpAddr - 1
7
8      if (rv ∉ tmpAddr->setVar) then
9              result->addr = null
10             // rv has not been assigned yet
11             return
12     else
13             result->addr = tmpAddr
14 else
15     tmpAddr = ba - 1
16     while (tmpAddr ≥ loopbody->startAddr)
17               ∧ (rv ∉ tmpAddr->setVar) do
18                tmpAddr = tmpAddr-1
19     if (rv ∉ tmpAddr->setVar) do
20           tmpAddr = loopbody->endAddr
21           while (tmpAddr≥loopbody->startAddr)
22                ∧ (rv ∉ tmpAddr->setVar) do
23                   tmpAddr = tmpAddr - 1
24     result->addr = tmpAddr
25
26 assAddr = result->addr
27
28 if assAddr > ba then
29     result->inst = i + assAddr->rv->offset
30 else
31     result->inst = i - 1 + assAddr->rv->offset
```

Figure 11: ba is in loopbody

7.2.2 *Loop Body*

If ba is in the loop body, we have to look for the address assigned rv in a slightly different manner. Indeed, if the break occurred during the first iteration of the loop body, then it might be an address of the prologue.

Once we know where rv has been assigned, thanks to the offset, we can tell in which iteration it occurred.

7.2.3 *Epilogue*

The epilogue is very similar to the prologue, except that we count the number of times rv has been assigned from the end, not from the beginning.

```
1  tmpAddr = ba - 1
2  while (tmpAddr ≥ prologue->startAddr)
3       ∧ (rv ∉ tmpAddr->setVar) do
4         tmpAddr = tmpAddr-1  // search backward for rv
5  result->addr = tmpAddr
6
7  assAddr = result->addr
8
9  result->inst = epilogue->lastIter
10 tmpAddr = epilogue->endAddr
11 while (tmpAddr ≠ assAddr) do
12        if (rv ∈ tmpAddr->setVar)
13       ∧ (tmpAddr->rv->lnum == assAddr->rv->lnum) then
14              result->inst−−
15        tmpAddr = tmpAddr - 1
```

Figure 12: ba is in epilogue

8. EXPERIMENTATION CONTEXT

8.1 Tool Framework

Our experimentation is done with a tool framework [14] including a compiler, MMDSP+ C Compiler, a debugger, MMDSP+ GDB, and a set of specific applications.

MMDSP+ C Compiler is a compiler for an embedded processor called MMDSP [5]. It performs state-of-the-art optimizations. It has been designed with a modular framework allowing the development and integration of new optimizers both high and low levels [5]. Its architecture is represented in figure 1. It is built around the CoSy compiler development suite [2]. The in-house back-end, named eliXir, is an STMicroelectronics proprietary back-end designed to replace CoSy back-end optimizers (e.g. the scheduler and register allocator) with a more modular infrastructure allowing more aggressive low-level optimizations (e.g. software-pipelining, peephole, post addressing mode). The structure of EliXir is represented in figure 13.

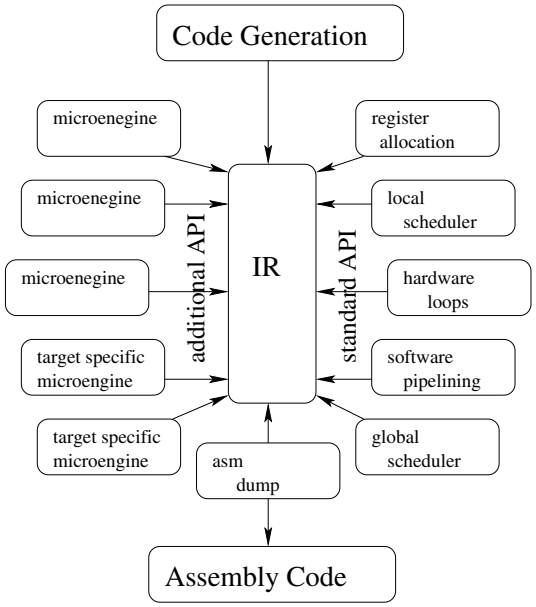

Figure 13: EliXir Structure

MMDSP+ GDB is based on the GNU Source-Level Debugger gdb [17]. It uses GDB/MI protocol to communicate with Eclipse[1].

[1]http://www.eclipse.org/

It has been adapted to handle gracefully the 24 bits memory of the MMDSP. It also handles circular pointers and the dynamic 16/24 bits arithmetic mode of the processor.

The set of applications contains several audio codecs such as the Enhanced Full Rate (EFR) codec of the GSM wireless communication standard, g723.1, mp3 and AMR codecs. 80% of the code is made of loops; `mmdspcc` performs software-pipelining on 30% to 50% of these loops, depending on the codec.

8.2 Experiment

To validate our approach, we divided our experiment into two steps. In the first step, we implemented and validated the approach presented in this paper with every other optimization turned off. In the second step, we instrumented selected engines of compiler so they kept up-to-date debugging information before and after the software-pipelining engine.

1^{st} Step- The first step is fully described in the present paper. Our instrumentation stores values previously computed by the original algorithm, hence it is less than 100 lines of code. We have extended the debugger with approximatively 900 lines of code: requested variables' values are now given with a context of execution, see figure 14. This example is written in a `gdb`-like syntax.

```
(gdb) break 7
 Breakpoint 1 at 0x33C and 0x348: file loop.c, line 7.
(gdb) run
 Starting program
 Reading symbols for shared libraries . done

 Breakpoint 1 at loop.c:7
 7                      i = i + 1;
 WARNING: software-pipelined loop:
          2 iterations overlapped
(gdb) continue 2
 Will ignore next 2 crossings of breakpoint 1.
 Continuing.

 Breakpoint 1 at loop.c:7
 7                      i = i + 1;
 WARNING: software-pipelined loop:
          2 iterations overlapped
(gdb) print x
 x = 6
 WARNING: software-pipelined loop:
          accurate for iteration (i == 3)
(gdb) print y
 y = 4
 WARNING: software-pipelined loop:
          accurate for iteration (i == 2)
```

Figure 14: For an example corresponding to the debugging of program in figure 7(a). This example follows the same steps of execution as example in figure 7.A breakpoint is set at line 7 and the program runs until reaching it.

2^{nd} Step- The second step is not described in this paper, we shall only give a brief description. Our algorithm supposes debugging information to be accurate before running the software-pipelining engine. In order to respect this assumption, we have instrumented MMDSP+ C Compiler so other engines maintain information up-to-date during compilation. We have followed a non-transparent approach. Labeling instructions through optimizations allows the debugger to map target instructions with source statements even though the source code has been highly optimized.

We restricted our field of investigation to four engines: **dead code** elimination, common subexpression elimination (**CSE**), **hardware loop** creation and miscellaneous **code motions** such as code sinking and code hoisting. Since we had to write a library to manipulate information in the previous step, the instrumentation corresponds to approximatively 1400 lines of codes and less than 100 lines of code for each engine. The set of selected optimizations is relevant because it covers every kind of modification optimizations can perform to the code: elimination (**dead code** elimination), duplication, rewriting (**CSE**, **hardware loop** creation), reordering (misc. **code motions**). It also includes modifications made to the instructions instances order and not only to instructions (see C.Jaramillo [13]).

So far, our implementation did not show any dysfunctions. Notice that there is not any relevant metric to demonstrate the validity of our approach because a non-transparent debugger relies on the user. The size of debugging information added to the binary file is not a significant information and the time needed by the debugger to load it is negligible.

9. PREVIOUS WORK

To the best of our knowledge, debugging software-pipelined loops has never been addressed non-intrusively or without specific hardware mechanisms.

John Hennessy [12] was the first to define notions of non-current variables and endangered variables. If an optimized program aborts or a debugger stops the program, the resulting value of the variable may not equal the value of the variable in the original source program at the corresponding point, and is called *non-current*. A variable is called *endangered* when its currency depends on the path taken. Hennessy's ideas have served as the foundation for much of the research done in this area. Wall, Srivastava, and Templin [21] and Copperman and McDowell [7] later revised Hennessy's original algorithms to correct for changes in compiler technology that invalidated some of Hennessy's original assumptions.

Polle T. Zellweger [26, 25] focused on two optimizations: cross-jumping, an optimization which consists in replacing a portion of code by a function call, and inlining, an optimization which consists in replacing a function call by a copy of the function itself. The solution she proposed did not address the debugging of software pipelined loops or instruction scheduling, but she gave original definitions for **transparent behavior**, and correct, or non-transparent behavior. Indeed, a debugger is said to have a transparent behavior when its responses to user requests concerning the execution of an optimized program are the same as responses would be for an unoptimized version of the program. A debugger is non-transparent, or correct, when it can display, in source program terms, the relevant changes caused by optimization at execution point

Coutant, Meloy and Ruscetta proposed what they called *A Practical Approach To Source-Level Debugging of Globally Optimized Code* [8]. Their tool, DOC, is a prototype of the existing C compiler and source-level debugger for HP900 Series 800 which deals almost exclusively with the data value problem caused by optimizations. Their solution partially solves the data value problem trying to remain consistent with the user's perception of the source, i.e. they try to recover variables values.

Brooks, Hansen and Simmons' work [6] aims at providing a visual feedback during the debugging of optimized code. They designed the Convex debugger, CXdb. It follows a non-transparent approach: it highlights various expressions within the source statements as the corresponding instructions are executed. No explanation is provided to the user, thus, unless the latter knows which optimizations might have been performed, it can become very defficult to determine what happened, and hence understand the context. While they solved the location problem, they only partially addressed data value problems. A resident variable is implicitly

suspect - it is up to the user to determine how the runtime value of a variable relates to the source value. Even though this approach is easily maintainable due to its simple labeling scheme, it provides only a cursory solution for data value problems.

Roland Wismüller [22] proposed an algorithm for solving the currency problem. Based on data flow analysis, his solution determines whether a variable's value is current or not, but no solution is provided to recover any value. His algorithm is based on a comparison of two data flow graphs. They both are unrolled in order to make the difference between two instances of the same instruction. The same unrolling technique has been used by Dhamdhere and Sankaranarayanan [9]. Wismüller is also the first to use a mapping which uses *semantically equivalent* instructions, what C.Tice will later call *key instructions* [18].

J.Gough, J.Ledermann and K.Elms [11] described an alternative model in which any needed information is extracted by coopting a modified version of the compiler during the debugging session. They do not explicitly solve the code location problem nor do they solve data value problems. Three years later, K.Elms [10] released a more detailed implementation.

Ali-Reza Adl-Tabatabai [3] addressed the code location problem and data problems for a given set of scalar optimizations. His code location problem solution relies on a dual labeling method and expects breakpoints and exceptions to happen in the expected order if instructions are not scheduled. In order to track modification brought to statements, the compiler maintains two mappings: the object-to-source mapping and the source-to-object mapping. As stated in his conclusions [3, page 171], he does not address the debugging of software-pipelined loops.

Le-Chen Wu and Wen-Mei W.Hwu [23, 24] address the code location problem from the user's point of view. They proposed an implementation scheme for setting breakpoints. They implemented their technique by modifying an existing compiler and running tests with it. They concluded that their scheme was unable to handle loop transformations which reordered instructions from different iterations of the original loop in the same iteration of the new loop (software pipelining, for instance).

D.M.Dhamdhere and K.V.Sankaranarayanan [9] presented a partial solution to the data value problem. Their proposal is based on a neat reduction of the data flow graph into a list of nodes ordered according to the time when their last occurrence was executed. The data flow graph is unrolled and then reduced to a list. To do so, they used the Zellweger's time-stamp idea [26] and proved the equivalence of the two graphs under hypothesis of restriction. They concluded that their proposal is a practical basis for providing partially transparent debugging.

Caroline Tice's approach to the problem is based on the idea that programmers may want to know what happened to their code during optimizations [20]. She developed a tool, OPTVIEW [19], which displays parts of optimizations to the user. Her tool displays a modified version of the source code, with comments added whenever possible. Her main contribution is the *key instruction* concept which allows the debugger to set source-level breakpoints. In order for this mechanism to be accurate, for each source statement she defines which instruction embodies most closely that statement's semantics. For example, the key instruction for an assignment statement is the one that stores the assignment value either to the variable's location in memory, or to a register (if the write to memory has been eliminated). The key instruction for a function call is the jump to the code for that function. She also addresses the eviction recovery of wandering data by modifying Coutant, Meloy and Ruscetta paper's scheme of variable range table. She proposed setting hidden breakpoints to recover data, but only in subroutines where the user already had set breakpoints in order to minimize the slow down of execution. In her conclusions [20], she considers the limitations of her work especially as it applies to loop transformations. She provides a summary of user feedback, but her tool is not available for experimentation.

C.Jaramillo [13] proposed a fully transparent solution to symbolic debugging of optimized code. Her main goal is to hide every single transformation from the user by making heavy use of hidden breakpoints and tables such as reportable variables table, overwritten variables table, range table, and many others. As a result, it seems to drastically slow down the program execution during the debugging session. We believe it is a major showstopper to industrial usage of these technics.

10. CONCLUSION AND FURTHER WORK

This paper addresses the non-transparent debugging of software-pipelined loops. We proposed an algorithm for the debugger to reveal non-transparently the program's behavior using information stored by the compiler while software-pipelining a program. This approach only depends on the scheduling algorithm and not on the processor. We also presented our experimentation context based on an embedded system processor: the MMDSP which implements a modulo-software-pipelining algorithm. To the best of our knowledge, this problem has never been addressed before. Our development framework prototype needs the developer to already know a bit about compilation optimizations. This may be seen as a disadvantage while it actually is our intention not to hide details to the programmer. The embedded developers who will be our tool's first class users usually have sufficient knowledge of their systems to understand the additional information we provide.

As further work, we will extend our method to other sofware-pipelining algorithms, so that compilers which implement them could be extended as well. The cornerstone of this work will be showing that the algorithm respects property 5.3.

Another further work would be trying to adapt our algorithm to a transparent debugging of software-pipelined loops. This would be the exact opposite approach to the one adopted here. Using invisible breakpoints, slicing techniques, and pools to store values are promising ideas, and many papers on the subject of transparent debugging are built on them.

11. ACKNOWLEDGMENTS

The authors would like to thank Vincent Lorquet, Quentin Clombet, Vincent Colin-de-Verdière, Jean-Marc Daveau and Thierry Lepley from the CEC team for their help during our experiments. The authors would also tlike to thank Laurent Gérard and Denis Pilat from the IDTEC Team for their helpful comments.

12. REFERENCES

[1] A WORKGROUP OF THE FREE STANDARDS GROUP. *The DWARF Debugging Standard.* http://dwarf.freestandards.org/.

[2] ACE ASSOCIATED COMPILER EXPERTS BV. CoSy Compilers, Overview of Construction and Operation. White paper, 24 Apr. 2003.

[3] ADL-TABATABAI, A.-R. *Source Level Debugging of Globally Optimized Code*. PhD thesis, Carnegie Mellon University, Pittsburgh PA 15213-3891, June 1996.

[4] APPEL, A. W. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[5] BERTIN, V., DAVEAU, J.-M., GUILLAUME, P., LEPLEY, T., PILAT, D., RICHARD, C., SANTANA, M., AND THERY,

T. Flexcc2: An optimizing retargetable c compiler for dsp processors. In *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software* (London, UK, 2002), Springer-Verlag, pp. 382–398.

[6] BROOKS, G., HANSEN, G. J., AND SIMMONS, S. A new approach to debugging optimized code. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation* (New York, NY, USA, July 1992), vol. 27, ACM Press, pp. 1–11.

[7] COPPERMAN, M., AND MCDOWELL, C. E. A further note on hennessy's "symbolic debugging of optimized code". *ACM Trans. Program. Lang. Syst. 15*, 2 (1993), 357–365.

[8] COUTANT, D. S., MELOY, S., AND RUSCETTA, M. Doc: a practical approach to source-level debugging of globally optimized code. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation* (New York, NY, USA, July 1988), vol. 23, ACM Press, pp. 125–134.

[9] DHAMDHERE, D. M., AND SANKARANARAYANAN, K. V. Dynamic currency determination in optimized programs. *ACM Trans. Program. Lang. Syst. 20*, 6 (1998), 1111–1130.

[10] ELMS, K. Debugging optimised code using function interpretation. In *Automated and Algorithmic Debugging* (1997), pp. 27–36.

[11] GOUGH, K., LEDERMANN, J., AND ELMS, K. Interpretive debugging of optimised code. *Proceedings of ACSC-17, Christchurch* (1994).

[12] HENNESSY, J. Symbolic debugging of optimized code. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 323–344.

[13] JARAMILLO, C. I. *Source Level Debugging Techniques And Tools For Optimized Code*. PhD thesis, University of Pittsburgh, 2000.

[14] PAULIN, P. G., AND SANTANA, M. Flexware: A retargetable embedded-software development environment. *IEEE Des. Test 19*, 4 (2002), 59–69.

[15] RAU, B. R. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture* (New York, NY, USA, 1994), ACM Press, pp. 63–74.

[16] STALLMAN, R. M. *Using and Porting the GNU Compiler Collection, For GCC Version 2.95*. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, 1999.

[17] STALLMAN, R. M., PESCH, R., SHEBS, S., ET AL. *Debugging with GDB: The GNU Source-Level Debugger*. 2002.

[18] TICE, C., AND GRAHAM, S. *Key Instructions: Solving the Code Location Problem for Optimized Code*. Compaq, Systems Research Center, 2000.

[19] TICE, C., AND GRAHAM, S. L. Optview: A new approach for examining optimized code. In *Workshop on Program Analysis For Software Tools and Engineering* (1998), pp. 19–26.

[20] TICE, C. M. Non-transparent debugging of optimized code. Tech. rep., University of California at Berkeley, Berkeley, CA, USA, November 1999.

[21] WALL, D., SRIVASTAVA, A., AND TEMPLIN, F. A note on hennessy's "symbolic debugging of optimized code". *ACM Trans. Program. Lang. Syst. 7*, 1 (January 1985), 176–181.

[22] WISMÜLLER, R. Debugging of globally optimized programs using data flow analysis. *ACM SIGPLAN Notices 29*, 6 (1994), 278–289.

[23] WU, L.-C., AND MEI W. WHU, W. A new breakpoint implementation scheme for debugging globally optimised code. *Urbana 51* (1998), 61801.

[24] WU, L.-C., MIRANI, R., PATIL, H., OLSEN, B., AND MEI W. HWU, W. A new framework for debugging globally optimized code. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation* (New York, NY, USA, 1999), ACM Press, pp. 181–191.

[25] ZELLWEGER, P. T. An interactive high-level debugger for control-flow optimized programs. In *SIGSOFT '83: Proceedings of the symposium on High-level debugging* (New York, NY, USA, August 1983), vol. 18, ACM Press, pp. 159–172.

[26] ZELLWEGER, P. T. *Interactive source-level debugging for optimized programs (compilation, high-level)*. PhD thesis, 1984.