MHAOTEU: Memory Hierarchy Analysis and Optimization Tools for The End-User

REPORT M3.D1

Advanced Performance Analysis

Jaume Abella, Grigori Fursin, Antonio Gonzalez, Josep Llosa, Mike O'Boyle,
Abhishek Prabhat, Olivier Temam, Sid Ahmed Ali Touati,
Xavier Vera, Gregory Watts

February 12, 2001

Abstract

This deliverable contains :

- Static Locality Analysis, by Jaume Abella (UPC), Antonio Gonzalez (UPC), Josep Llosa (UPC), Xavier Vera (UPC[1]) .
  Modifications made in UPC's CME implementation in order to perform some driven optimizations.

- A Fast and Accurate Evaluation of a Memory Performance Upper-Bound, by Grigori Fursin, Mike O'Boyle (University of Edinburgh, UK), and Olivier Temam, Gregory Watts (Paris South University, France).
  Determining the potential benefit of hand optimisation is crucial for large applications where the effort involved may be possibly many man months. This deliverable develops a new technique which provides an upper-bound on memory performance based on assembler code modification. By translating memory references so that they always hit the cache and simultaneously preserving register dependences, we develop a tool that can provide an upper bound on memory behaviour several without the need for slow simulation. This tool has been applied to the Spec95 benchmark swm, where it shows that the upper bound on the code's performance is approximately a half of the original execution time.

- Definition of Optimality, by Sid Ahmed Ali Touati (INRIA).
  Tools for computing the performance of a loop on processors.

- Links between source level and assembly level, by Abhishek Prabhat (INRIA[2]) :
  - Mapping between source and assembler with ATAC;
  - Debugging Optimized Code: a survey.

[1] currently at MdH (Västerås, Sweden)
[2] currently at Indian Institute of Technology, Delhi, India

Contents

List of Figures

Part I
Static Locality Analysis

Jaume Abella (UPC), Antonio Gonzalez (UPC), Josep Llosa (UPC), Xavier Vera (UPC)

1 CME

In this section we describe the modifications made in our CME implementation in order to perform some driven optimizations.

1.1 Overview of Cache Miss Equations

Cache Miss Equations are a set of equations[1] that represent all the potential cache misses for the references in a loop nest. They describe the precise relationship among the iteration space, array sizes, base addresses, and the cache parameters for a loop nest. This section presents an overview of the CMEs. For more details about CMEs see [4, 5].

```
parameter (N)
REAL a(N,N), b(N,N), c(N,N)
do i = 1, N
  do j = 1, N
    do k = 1, N
      a(i,j) = a(i,j) + b(i,k) * c(k,j)
    enddo
  enddo
enddo
```

Figure 1: Matrix multiply algorithm

In order to generate CMEs, the *reuse vectors* [8] of all the references in a loop nest must be generated. *Reuse vectors* provide information about the potential reuses in the entire iteration space. Figure 1 shows the *matrix multiply* kernel. For instance, $\vec{r} = (0, 0, 1)$ is a reuse vector for reference *c(k,j)*, because the data accessed by this reference is potentially reused one iteration later (it is potentially mapped into the same cache line). In order to determine if these potential reuses are realized, CMEs must be generated and solved.

For every reuse vector of a reference two types of CMEs are generated:

- **Compulsory equations.** Compulsory equations represent the first time a memory line is brought into the cache. We may distinguish between two groups of equations:
  - **Cold Miss Equations** are associated to every spatial or temporal reuse vector. They describe the iteration points where the reuse is not realized because the studied reference reuses from an iteration point outside the iteration space.

[1] The term equation has been loosely used to refer to a set of simultaneous equalities and inequalities.

  - **Cold Miss Bounds** are associated to every spatial reuse vector. They describe the iteration points where the reuse does not hold because the potential reused data is in fact in a different cache line.

- **Replacement equations.** Given a reference, replacement equations represent the interferences with any other reference. For each pair of references (R_A and R_B), the following expression gives the condition that determines whether they are mapped onto the same cache set:

$$\begin{aligned} Cache\_Set(\vec{i})_{R_A} &= Cache\_Set(\vec{j})_{R_B} \\ \vec{j} &\in \mathcal{I} \end{aligned}$$

  where $\mathcal{I}$ represents the iteration points between $\vec{i}$ (the current one) and the iteration point from which R_A reuses. This condition is expanded into a set of equations for each reuse vector.

1.2 Finding Cache Misses from CMEs

Deciding whether a reference causes a miss or a hit for a given iteration point is equivalent to deciding whether this iteration point belongs to the polyhedra defined by the CMEs. The points inside each CMEs polyhedron represent the potential cache misses (the number of points is the number of potential cache misses). This leads us to consider several ways for computing them:

- **Solver.** Given a reference R with m reuse vectors and n_k equations for the k^{th} reuse vector, the polyhedron that contains all the iteration points that result in a miss is [4]:

$$Set\_Misses = \cap_{k=1}^{m} \cup_{j=1}^{n_k} Solution\_Set\_Equation_j$$

  This approach implies to count the number of points inside the union of convex polyhedra. This requires counting the points (which is a NP problem for a single polyhedron) into an exponential number of polyhedra, making this problem infeasible due to its huge computing time.

- **Traversing the iteration space.** Given a reference, all the iteration points can be tested independently [5]. In order to know if an iteration point $\vec{i}_0$ results in a miss we need to know when it fulfills the CMEs. This problem is equivalent to finding out whether, after substituting the iteration point in the CMEs, the resulting polyhedron is non-empty. This is still a NP problem, but only a linear number of polyhedra must be analyzed for each iteration point. However, the problem is still infeasible except for tiny iteration spaces.

Moreover, in a k-way set associative cache, there are k cache lines in every set, so k distinct contentions are needed before a cache miss occur. Therefore, the first method can only be applied to direct-mapped caches whereas the second method works for both direct-mapped and set-associative organizations.

1.3 A Fast and Accurate Implementation to Solve CMEs

In this section we describe a fast and accurate approach to estimate the solution to the CMEs. Our approach builds upon the second method to solve the CMEs (traversing the iteration space). This approach to solve the CMEs allows studying each reference in a particular iteration point independently of all other memory references and iteration points.

We have developed some techniques that exploit the special characteristics of the CMEs polyhedra [1] in order to speed-up the process of counting points in polyhedra:

- **Counting Compulsory Polyhedra.** When an iteration point is substituted, Compulsory Equations result in polyhedra with either 0 or 1 variable. A polyhedron with 0 variables consists of a set of inequality relations between integer values. The iteration point that has been substituted is a potential miss if the inequalities hold. On the other hand, since a polyhedron with 1 variable represents an interval, the iteration point $\vec{i}_0$ results in a potential miss when there exist integer values in it.

- **Counting Replacement Polyhedra.** Due to the particular form of these polyhedra, the number of integer solutions can be computed in a more efficient way than in general polyhedra. We have developed a method to detect when they are empty that is based on counting the number of integer points inside of them [2]. In order to compute it, the domains[2] of the different variables involved in its definition are calculated. This can be done by means of the vertices of the polyhedron, but computing them is an NP problem. We have developed specific techniques for replacement polyhedra that compute the domains of the variables in a polynomial time. A detailed description of these techniques can be found in [2].

The use of these techniques results in an average speed-up of 20 over a method based on identifying the vertices of the polyhedra. However, this important speed-up is still insufficient to solve CMEs for huge iteration spaces in a reasonable amount of time.

To further reduce the computation cost, we propose to study a subset of the iteration space instead of the whole iteration space [7]. As outlined above, CMEs allow us to study each iteration point independently of all other iteration points. Based on this property we can generate the subset of points using *Simple Random Sampling* [6].

We model the number of misses of each reference using a Discrete Random Variable. This random variable follows a Binomial distribution that models phenomena consisting of n different and independent experiments that follow a Bernoulli distribution. Therefore we can use statistical techniques [3] in order to compute the parameters that describe this random variable. The approach to obtain an approximation of the miss ratio is to evaluate the behavior of a subset of the population (sample) obtaining the empiric value of the parameters that describe the sample and to infer these values to the population.

The size of the sample is set according to the required width of the confidence interval and the desired confidence. We found experimentally that a confidence interval of width 0.05 and a 95% confidence is enough to obtain accurate miss ratios with a very small computing time (only 1082 points of the iteration space must be explored). In this way,

[2]We define the real domain of a variable x in a polyhedron P as the range of real values it takes inside P.

the miss ratio is computed as an interval of width 0.05 and the actual miss ratio belongs to this interval with a probability of 95%. The central point of this interval can be used as an estimation of the actual miss ratio.

Using these parameters, we have evaluated the miss ratio of the main loops of the SPECfp95 programs. The results show that the absolute error in the miss ratio is smaller than 0.002 in 65% of the loops and never greater than 0.01. In addition, the analysis time for each program is usually just a few seconds.

2 Improving CME implementation

CME for multiple convex regions

The implemented technique to count points in polyhedra requires that the iteration space is a convex region, but after tiling n dimensions the iteration space is the union of 2^n convex regions.

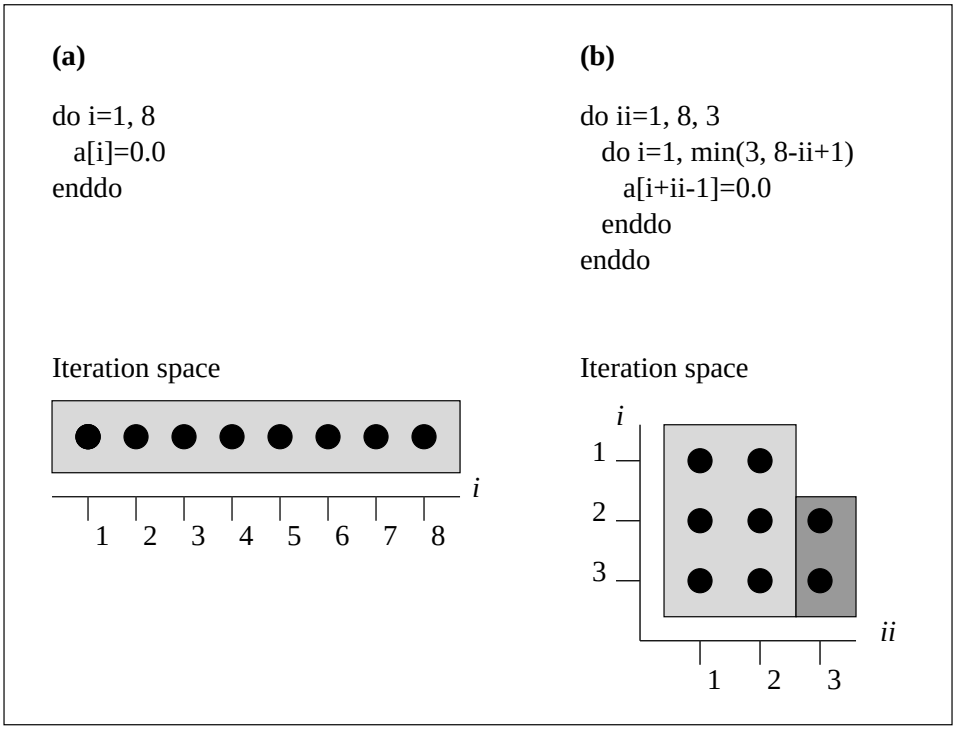


Figure 2: Example of iteration space: (a) before tiling, (b) after tiling

Figure 2 shows how the iteration space of a one-dimension loop becomes a two-convex region iteration space after tiling. The shaded regions correspond to the different convex regions before and after tiling.

There are different ways to solve this problem. The easiest option consists of defining only one convex region that approximates the actual non-convex region. This convex region can be the smallest parallelepiped that includes all other convex regions (see figure 3 (a)) or the region which does not include the last iteration of every tiled loop where the tile

size is not a divisor of the upper bound (see figure 3 (b)). Both options have drawbacks. The first option includes in the convex regions points outside the iteration space, while the second option does not include points belonging to the iteration space.

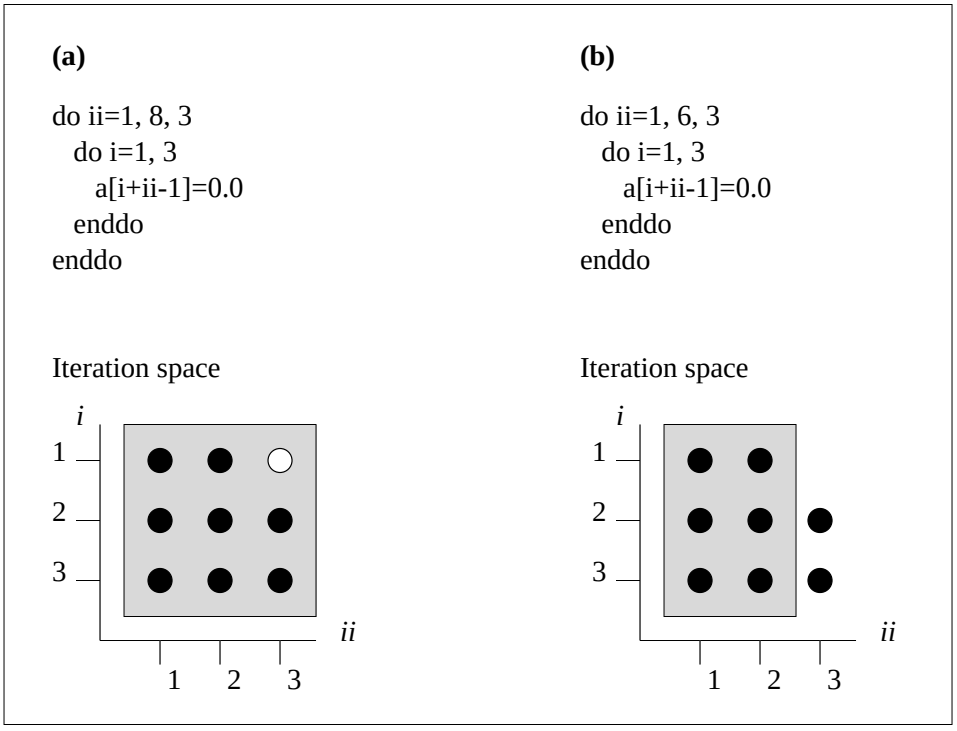


Figure 3: Convex regions approximations: (a) bigger region, (b) smaller region

Because of this, we have decided to implement a more accurate solution. The CME implementation has been modified to deal with multiple convex region by defining the equations for every convex region and solving for every analyzed point the equations corresponding to the convex region in which the point is contained. Let n be the number of convex regions of a loop after tiling. Every compulsory equation should be defined for each convex region, so the number of compulsory equations is increased by a factor of n. For each reuse vector, we have to generate a set of replacement equations for each convex region. In addition, we have to generatre a set of equations for every pair of convex regions that reflect the potential reuse between different regions. Due to this, the number of replacement equations is increased by a factor of n^2.

References

[1] Nerina Bermudo, Xavier Vera, Antonio González, Josep Llosa. An efficient solver for cache miss equations. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2000.

[2] Nerina Bermudo, Xavier Vera, Antonio González, Josep Llosa. Optimizing cache miss equations polyhedra. In *4th Workshop on Interaction between Compilers and Computer Arquitecture (Interact)*, 2000.

[3] M.H. DeGroot. *Probability and statistics.* Addison-Wesley, 1998.

[4] Somnath Ghosh, Margaret Martonosi, Sharad Malik. Cache miss equations: an analytical representation of cache misses. In *ICS97*, 1997.

[5] Somnath Ghosh, Margaret Martonosi,d and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *ASPLOS98*, 1998.

[6] David S. Moore, George P. McCabe. *Introduction to the Practice of Statistics.* Freeman & Co, 1993 (2nd edition).

[7] Xavier Vera, Josep Llosa, Antonio González, Carlos Ciuraneta. A fast implementation of cache miss equations. In *8th International Workshop on Compilers for Parallel Computers (CPC)*, 2000.

[8] Michael E. Wolf, Monica S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN91*, 1991.

Part II
A Fast and Accurate Evaluation of a Memory Performance Upper-Bound

Grigori Fursin, Mike O'Boyle (University of Edinburgh, UK), and Olivier Temam, Gregory Watts (Paris South University, France)

1 Introduction

The potential benefits of memory optimizations are numerous, but one of the characteristics of memory optimizations is that the benefits vary wildly. Depending on application domain, programming style and program parameters, optimizing a program memory-wise can bring small or large performance improvements. While these variations are not an issue as long as program optimization is performed in an academic environment, companies may have a harder time coping with them. An industrial size program can include several hundred thousands lines and optimizing such a program can take weeks or months. In order to accept to invest the corresponding number of person months, a company must get a hint at the potential performance improvement that these optimizations (this investment) might bring. This rather practical and economical problem then turns into a rather tricky scientific problem: we must find a way to estimate the program execution time assuming we have optimized its memory behavior. While it is difficult to provide an accurate value of the execution time, we can seek a lower bound of the execution time. Besides estimating the potential benefit of program optimization, this lower bound can also be used to *drive* an iterative optimization process by comparing the optimized program with the lower bound at each step, and deciding whether the remaining potential performance gains are worth an additional effort or whether the optimization process can stop.

In this deliverable, we propose a method for computing the execution time of a program assuming all misses have been removed. Processor simulation where the cache is perfect, i.e., all cache accesses hit, can be used to provide this estimate but the method has many flaws. First of all, it is difficult to develop a simulator that closely mimics the performance of an existing processor because undisclosed architecture design adjustments can have a profound impact on processor performance. Second, programs where memory performance is critical exercise not only the processor architecture but many other components that are located outside the chip: the bus chipset (bus management), the lower caches, the memory boards and the operating system for memory management in some cases (handling page table entries), not to mention the interactions between the operating system and the program (in the processor and in the different memory levels), the physical mapping of lines in lower cache levels and so on. So even though processor simulation can provide trends, it is likely to be very inaccurate unless a full system simulator is provided by the processor manufacturer. Whether one has a processor simulator or a detailed full system simulator, the high simulator slow down factors (500 to 1000 for a simple processor simulator) will not make them practical tools for quickly evaluating the potential benefit

of program optimizations, and even less so for evaluating the remaining potential benefit at each step of an optimization process. Program optimization is a trial and error process that requires constant program modification and testing, and in an industrial environment, it is not always compatible with tools having an excessively long response time.

So the challenge is to develop a method that is both fast and reasonably accurate. The principle of our method is to transform array references into scalar references and still manage to execute exactly the same instructions (same number of instructions, same instruction types). Then, the memory footprint of the whole program becomes extremely small inducing almost no miss, so that the instrumented program execution time is almost exactly that of the original program but without any miss.

For that purpose, we consider each procedure and each loop in a procedure. For each loop, we analyse the assembly code generated by the production compiler and identify the instructions where the addresses of array references are incremented. Then we inhibit the address modification by changing the address increment to 0. With this technique, *all* instructions remain *identical*, except that a parameter of some instructions changes. Since the modifications only affect constants, not only the transformed code has exactly the same instructions, but all register dependences remain identical. This latter point is critical for new out-of-order execution processors where register dependences strongly affect when instructions are issued and the overall processor performance.

2 Example

On the Alpha EV6 for instance, the address incrementation of an array reference looks as follows:

```
lda $19, 8($19)
........
ldt $f13, ($19)
```

Register `19` contains the address of a load instruction (`ldt`), and it is incremented by 8 on each loop iteration within instruction `lda` (the acronym is misleading, it is an add instruction dedicated to address computation). Assume now that we modify the first instruction as follows:

```
lda $19, 0($19)
........
ldt $f13, ($19)
```

Only the offset of the `lda` instruction has been modified: the 8 has been replaced with a 0. Now, the instructions are the same, the same number of computations is performed and the register dependences have not been modified. But the main point is that the address referenced by instruction `ldt` *always remains the same* through the whole loop execution. So, if we perform this transformation for all array references in a loop, we obtain a loop with references to scalars instead of references to arrays. Therefore, the memory footprint of the loop is equal to the number of array references (plus the original scalar references which are not affected), and this number is very small compared to the cache size in most cases. Consequently, the number of cache misses of the corresponding loop is very small.

Below, we present an example for a complete loop in SWIM. We first present the original source and assembly code and then the transformed assembly code. The modifications in the assembly code are outlined.

Source loop:

```
      DO 215 I=1,M
      UNEW(I+1,N+1) = UNEW(I+1,1)
      VNEW(I,1) = VNEW(I,N+1)
      PNEW(I,N+1) = PNEW(I,1)
  215 CONTINUE
```

Original assembly code:

```
 #    335       DO 215 I=1,M
 #    336       UNEW(I+1,N+1) = UNEW(I+1,1)
s4subq $0, $0, $25
mov 1, $17
s4subq $25, $0, $25
lda $20, 8($3)
s8addq $25, $0, $0
mov $22, $27
s8addq $0, $31, $0
mov $24, $2
s4subq $0, $0, $0
s4addq $0, $0, $0
ble $1, L$52
 #    337       VNEW(I,1) = VNEW(I,N+1)
lda $21, 10688($0)
lda $18, 10696($0)
addq $22, $21, $19
addq $3, $18, $18
 #    338       PNEW(I,N+1) = PNEW(I,1)
addq $24, $21, $21
lda $18, -10688($18)
lda $19, -10688($19)
lda $21, -10688($21)
L$53:
ldt $f20, ($20)
ldt $f13, ($19)
ldt $f18, ($2)
addl $17, 1, $17
lda $20, 8($20)
lda $18, 8($18)
lda $19, 8($19)
lda $27, 8($27)
lda $2, 8($2)
lda $21, 8($21)
```

```
cmple $17, $1, $23
stt $f20, -8($18)
stt $f13, -8($27)
stt $f18, -8($21)
bne $23, L$53
 #    339   215 CONTINUE
```

Transformed assembly code:

```
#    334   210 CONTINUE
 #    335       DO 215 I=1,M
 #    336       UNEW(I+1,N+1) = UNEW(I+1,1)
        s4subq        $0, $0, $25
        mov        1, $17
        s4subq        $25, $0, $25
        lda        $20, 8($3)
        s8addq        $25, $0, $0
        mov        $22, $27
        s8addq        $0, $31, $0
        mov        $24, $2
        s4subq        $0, $0, $0
        s4addq        $0, $0, $0
        ble        $1, L$52
 #    337       VNEW(I,1) = VNEW(I,N+1)
        lda        $21, 10688($0)
        lda        $18, 10696($0)
        addq        $22, $21, $19
addq        $3, $18, $18
 #    338       PNEW(I,N+1) = PNEW(I,1)
        addq        $24, $21, $21
        lda        $18, -10688($18)
        lda        $19, -10688($19)
        lda        $21, -10688($21)
        unop
L$53:
        ldt        $f20, ($20)
        ldt        $f13, ($19)
        ldt        $f18, ($2)
        addl        $17, 1, $17
MODIF: lda        $20, 0($20))
MODIF: lda        $18, 0($18))
MODIF: lda        $19, 0($19))
MODIF: lda        $27, 0($27))
MODIF: lda        $2, 0($2))
MODIF: lda        $21, 0($21))
        cmple        $17, $1, $23
        stt        $f20,-8($18)
```

```
stt         $f13,-8($27)
stt         $f18,-8($21)
bne         $23, L$53
```

The main remaining issue is to guarantee normal program execution after the transformation. If we perform this transformation on a loop, the whole program will be affected and it will obviously perform incorrectly. To circumvent that issue, we make a copy of each procedure in the program that contains a loop, and we instrument only the copy. Then we add a *restore* epilog after the instrumented procedure, to restore the few array references that were modified. The restore code calls the normal procedure, so that, in the end, we execute both the instrumented code and the original code.

Then, we profile the whole code and we deduce the optimal execution time from the cumulated execution time of all the *instrumented* procedures only. This result corresponds to the program execution time assuming all array references in program loops have been converted to references to scalars, which is very close to the execution of the same program where all references hit.

3 Implementation in a compiler

The instrumentation techniques (offset modification and restore code) described in this deliverable can be easily inserted in a production compiler and made available to all users through a flag. Still, in order to demonstrate our techniques on a large number of applications, we have started to develop a semi-automatic tool called *mpc* for performing these transformations on Alpha assembly code. It has been written in Java and only works with Fortran source codes. Ultimately, we would like to integrate it with the MHAOTEU framework.

The tool includes the following steps:

- duplicating subroutines to be instrumented:
- saving data at the beginning of the subroutine and restoring it at the end to preserve the control-flow dependencies of the whole application;
- detecting loops inside subroutine and detecting memory accesses which depends on the loop iterator;
- changing memory access computations so the addresses remain constant and don't depend on the loop iterator;
- rearranging memory accesses according to the cache L1 architecture in such a way that there is no conflict miss between the different (constant) array references inside the instrumented loop;
- running the instrumented application with profiling and deduce the optimal execution time (no miss).

4 Experiments

We have applied this technique successfully to a few SpecFP2000 programs. Below is the time profile of the original SWIM code.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
35.5	15.86	15.86	1	15860.35	44673.83	shalow_ [1]
26.0	27.49	11.63	200	58.16	58.16	calc2_ [3]
20.3	36.58	9.09	198	45.90	45.90	calc3_ [4]
17.9	44.58	8.00	200	40.00	40.00	calc1_ [5]
0.1	44.64	0.06	1	55.66	55.66	inital_ [6]
0.1	44.67	0.04	1	37.11	37.11	calc3z_ [7]

Below is the time profile of the SWIM code with the *calc2* routine instrumented. The original *calc2* routine appears as `calc2_` and the transformed *calc2* routine appears as `calc2mpc`.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
21.3	20.92	20.92	200	104.62	126.57	calc2_ [3]
19.1	39.75	18.83	200	94.14	126.43	calc1_ [4]
17.9	57.40	17.64	198	89.11	104.66	calc3_ [5]
14.5	71.63	14.24	1	14236.33	98358.40	MAIN__ [1]
12.8	84.19	12.56	198	63.45	63.45	calc3fgg_ [6]
6.6	90.65	6.46	200	32.29	32.29	calc1mpc_ [7]
4.5	95.04	4.39	200	21.95	21.95	calc2mpc_ [8]
3.1	98.12	3.08	198	15.55	15.55	calc3mpc_ [9]
0.1	98.26	0.13	1	133.79	133.79	inital_ [10]
0.1	98.36	0.10	1	101.56	101.56	calc3z_ [11]

You can note that the execution time of the *calc2* routine decreases from 11.63 seconds to 4.39 seconds if all array references become hits. It obviously shows that the cache has a tremendous impact on performance and this result is obtained far quicker than through a cache simulator. Hardware counters [2, 7] would provide this information just as fast but they wouldn't *quantify* the performance gain that can be achieved if misses are removed.

5 Issues

The techniques mentioned above raise several issues that can sometimes complicate their implementation.

- With the Compaq Alpha compiler, the loop index, in the above examples, is usually distinct from the array reference register index. If the loop index were the same as the array reference register index, we would need an additional register to perform the same transformation since it is not possible to set the loop iteration increment as 0.

- The scope of the techniques is restricted to loops and array references. It still encompasses a large set of Fortran codes, including sparse codes with indirect array references (the increment of the index array would then be set to 0). However, if a loop body contains a branch instruction, especially a branch instruction that depends on array values, then the technique cannot be applied as is. One solution is to evaluate the probability distribution of the branch outcome and then modify the test so that the branch outcome is generated by a random variable with this probability instead of the original test. This solution can perform reasonably well but it is partly satisfactory since the instrumented code instructions are not strictly identical to those of the original code, as for the loop bodies without branch instructions. Fortunately, a significant share of program loops in Fortran programs do not contain conditional branch instructions that depend on a value computed within the loop body.

6 Related Work

There are several possible approaches to the problem of defining a program memory performance upper bound.

First, several studies [1, 4, 6, 13] rely on Belady's MIN algorithm [3] to find the best possible memory behaviour but these studies only target a single memory level and the associated architecture is capable of selectively load, place and discard words and thus does not correspond to a real-life architecture.

The second most frequent approach is based on simulation. There is a very large amount of effort on simulation technology in the micro-architecture community [8, 5], including in the MHAOTEU project where we have developed the cache profiler and the cache debugger [14]. In [9], Martonosi et al. attempt to speed up cache simulation using trace sampling and achieve reasonably accurate results. However, simulation only provides a restricted view of the whole system performance, and it often ignores the impact of the operating system. Besides, lots of information on the system architecture are often not available.

One of the main benefits of an upper bound is a stop criteria for iterative compilation [11]. Iterative approach is a technique that explores an compiler optimisation space based on feedback information rather than static analysis. One of the key challenges identified in [11] is the need to incorporate sensible means to reduce the search space. Models have been proposed as on means of evaluating the worth of multiple possible optimisation candidates in [10] and have been used as a means of discarding poor candidates in [12]. However, simply reducing the number of points to consider is secondary to determining whether it is worth continuing to investigate the optimisation space. Having a useful upper bound on expected program behaviour, allows the iterative compiler to determine whether it is worth investigating further. This will also be useful to the automatic library tuning community [15].

7 Conclusions and Future work

We have developed a technique for quickly evaluating the execution time of a program assuming most cache misses have been removed. The execution time upper bound is fairly accurate as the program is not modified and is actually run on the machine studied. Thus all system and architecture artefacts are taken into account. The technique is way faster than simulation since the instrumented program's execution time is at most double the execution time of the original program versus a 500+ slowdown for simulation techniques. Our goal is to have this technique implemented in a production compiler to assist users who whish to evaluate the impact of cache performance on their program execution time and to evaluate the progress achieved at each step of an optimization process. The main limitation is the scope of the techniques (loops; loops with conditional branches are more difficult to handle). Another fruitful area is to incorporate this technique into an iterative compiler, which can trade off an execution budget against possible benefit.

While this technique can provide a program performance upper bound (a cache miss lower bound) it does not provide any hint at how this upper bound *can* be achieved, and even more generally, whether this upper bound can be achieved. Future research will focus on trying to define an upper bound that is close to what can be effectively achieved through classic program transformation techniques.

References

[1] Santosh G. Abraham, Rabin A. Sugumar. Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, Vol. 21-1, pp. 24-35, ACM Press, May 1993.

[2] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, William E. Weihl. Continuous profiling: Where have all the cycles gone? *IEEE Transactions on Computer Systems*, 15(4):357–390, November 1997.

[3] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[4] Douglas Burger, Alain Kägi, James R. Goodman. Memory Bandwidth Limitations of Future Microprocessors. In *Proceedings of the 23rd ACM International Symposium on Computer Architecture*, Philadelphia, pp. 78-89, May 1996.
`http://cardit.et.tudelft.nl/~steven/ilp/burger96.ps.gz`

[5] Douglas C. Burger, Todd M. Austin. The SimpleScalar Tool Set, version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.
`ftp://ftp.cs.utexas.edu/pub/dburger/papers/CAN_SS.ps`
`http://www.cs.wisc.edu/~mscalar/simplescalar.html`

[6] Douglas C. Burger, Alain Kägi, James R. Goodman. The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors. Technical Report 1261,

University of Wisconsin, Madison, April 1996.
`ftp://ftp.cs.wisc.edu/tech-reports/reports/95/tr1261.ps.Z`

[7] J. Dean, J. Hicks, C. A. Waldspurger, W. E. Weihl, G. Chrysos. ProfileMe: Hardware Support for Instruction Level Profiling on Out-of-Order. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, Research Triangle Park, North Carolina, November, 1997. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[8] Alvin R. Lebeck, David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer, vol. 27, no. 10, pp. 15-26*, October 1994.

[9] Margaret Martonosi, Anoop Gupta, Thomas Anderson. Effectiveness of trace sampling for performance debugging tools. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 248–259, Santa Clara, CA, May 1993. ACM SIGARCH.

[10] M.E. Wolf, D.E. Maydan, D.-K. Chen. Combining Loop Transformations Considering Caches and Scheduling. In *Int'l. J. of Parallel Programming*, pages 479–504, 1998.

[11] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, E. Rohou. Iterative Compilation in a Non-Linear Optimisation Space. In *Proceedings of the Workshop on Profile Directed Feedback-Compilation*, October 1998.

[12] P.M.W. Knijnenburg, T. Kisuki, M.F.P. O'Boyle. The Effect of Cache Models on Iterative Compilation for Combined Tiling and Unrolling. In *Proceedings of 3rd Workshop on Feedback-Directed and Dynamic Optimization Compilation*, December 2000.

[13] Olivier Temam. Investigating Optimal Local Memory Performance. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 1998.
ACM SIGPLAN Notices, 33(11), pp. 218-227, November 1998.

[14] Eric van der Deijl, Gerco Kanbier, Olivier Temam, Elena D. Granston. A Cache Visualization Tool. *IEEE Computer*, 30(7):71–78, July 1997.

[15] R.C. Whaley, J.J. Dongarra. Automatically Tuned Linear Algebra Software. *Available through* `http://www.netlib.org/atlas/`, 1998.

Part III
Definition of Optimality

Sid Ahmed Ali Touati, INRIA, France

1 Loop Performance Evaluation

When optimizing an application on a target architecture, we must chose a suitable metric for its performance. We think that simply using the absolute execution time in seconds is not significant since it does not take into account the processor frequency on which we execute. A program executing within one second on a processor of 800 MHZ has, in our point of view, a worse performance than if it executes within two seconds on the same processor but rated at 200 MHZ. The reason is that the first processor is theoretically four times faster than the second, but is executes the program only twice faster. The number of achieved processor clock cycles is a better metric since it enables us to compare the achieved performance regarding to the processor abilities. In the case of a loop (which is the purpose of the optimization process), it is better to scale this metric by the trip count and hence the loop performance unit becomes the number of achieved cycles per iteration. This section is devoted to present our tools which compute the performance of a loop.

1.1 Software Evaluation : timopl tool

Tool Description

Timpol is a tool which uses the processor timer to catch the absolute execution time of a loop. It determines the processor frequency statically by using an OS call. The performance of the loop is simply the execution time divided by the frequency and normalized by the trip count.

Timpol instruments the fortran application to get the value of the timer before entering and after exiting the loop. The following code shows an instrumented program to evaluate the performance of the innermost I-loop:

```
c ------------------------------------------------------------
c this file is automaticly generated by timopl  v 1.0
c          Touati Sid Ahmed Ali (INRIA)
c ------------------------------------------------------------
      program example
      ...

c timopl section: common variable declaration
      real  timopl_elapsed_time,timopl_before,timopl_after
      integer  timopl_nb_cycle
      logical  timopl_passed

c timopl makes some initializations
      timopl_passed = .false.
      do  k = 1,n
         do  j = 1,n
```

```
c timopl section: save the time before
            if (.not.(timopl_passed)) then
               call etime(timopl_before)
            endif
            do  i = 1,n
c           ..... fortran statements
            enddo

c timopl section: save the time after
            if (.not.(timopl_passed)) then
               call etime(timopl_after)
               timopl_passed = .true.
            endif
         enddo
      enddo
...
c timopl write results
      timopl_elapsed_time = timopl_after - timopl_before
c divide by the frequency
      timopl_nb_cycle = timopl_elapsed_time / 1.824818e-09
      print * , '---------------------------------------'
      print * , 'timopl results: Real Time in seconds: ', timopl_elapsed_time
      print * , 'timopl results: Real Cycle Number: ', timopl_nb_cycle
      print * , 'timopl results: Number Of Iterations', n - 1 + 1
      print * , 'timopl results: Cycle Per Iteration', timopl_nb_cycle /(n - 1 + 1)
      end
```

After the instrumentation, it executes the application n times ($n = 10$ by default). It compiles the application for all the compiler optimization levels (from none up to -O5). Finally, it builds a database of some statistics (min, max, mean, variance) of the achieved performances in each optimization level. If the performance is very different from one execution to another (high variance or important difference between the min and the max), the loop is sensitive to the interactions with the OS (process scheduling). The output of timopl is as follows:

```
number of executions for each compiler optimization level: 10

compiler optimization level : -O0
------------------------------
           min         max         mean                 variance
ExecTime   6.55055e-05 6.65486e-05 6.61644777777778e-05 1.06467786944446e-13
ExecCycle  21813       22160       22032.2222222222     11791.6944444444
CyclePerIt 222         226         224.222222222222     1.44444444444444

compiler optimization level : -O1
------------------------------
           min         max        mean                 variance
ExecTime   5.15133e-05 5.2914e-05 5.19553777777778e-05 1.89133276944445e-13
ExecCycle  17153       17620      17300.6666666667     21012.25
CyclePerIt 175         179        176.111111111111     1.86111111111111

compiler optimization level : -O2
------------------------------
           min         max        mean                 variance
ExecTime   2.62186e-05 2.7515e-05 2.70149888888889e-05 2.38061381111111e-13
ExecCycle  8730        9162       8995.33333333333     26417
CyclePerIt 89          93         91.4444444444444     3.02777777777778
```

```
compiler optimization level : -O3
------------------------------
           min         max         mean             variance
ExecTime   2.77311e-05 2.84836e-05 2.80235e-05      5.48826149999998e-14
ExecCycle  9234        9485        9331.44444444445 6121.27777777778
CyclePerIt 94          96          94.6666666666667 0.75

compiler optimization level : -O4
------------------------------
           min         max         mean                  variance
ExecTime   2.76119e-05 2.91727e-05 2.86189333333333e-05 2.32306995e-13
ExecCycle  9194        9714        9529.44444444445     25772.2777777778
CyclePerIt 93          99          96.5555555555556     3.02777777777778

compiler optimization level : -O5
------------------------------
           min        max         mean                  variance
ExecTime   2.8003e-05 2.90796e-05 2.85647111111111e-05 1.13188363611111e-13
ExecCycle  9325       9683        9511.55555555555     12543.5277777778
CyclePerIt 95         98          96.5555555555556     1.02777777777778
```

Usage

Timopl is called as follows:

```
ariane:~> timopl
        TOUATI Sid Ahmed Ali (INRIA 1998)
timopl: instruments loops in fortran programs for real performance evaluation
usage: timopl -i <inputfile>.f [-o<outputfile>.f] loop_specification -[x|c|r|h]
              [-f compile_option] [-p input_data_file] [-s <frequ>] [-n number_of_exec]

 input-file: file.f without .f, fortran file to analyze
 unparsed_file: instrumented file to generate (to stdout by default if -x option
                not used, else timopl.f)
 loop_specification: which loop to analyze. Two ways to do this :
        -l loop_line_number
        -[ marker. it is a string just before the loop,
           typically a user directive (comment).
 options:
        -x for compiling the fortran generated file and execute.
           Results are in timopl_results.
        -c for only compile (all fortran level optimization).
        -r for remove binary generated files.
 -f option to compile with.
 -p data input file to the program.
 -s to force the speed of the clock frequency in MHZ (an integer value).
    Automatic detection by default.
 -n number of execution. 10 by default
 -h host machine where to execute the experiments, while the instrumentation is done
    locally (it use rsh to do that, so verify your ~/.rhosts).
    You must specify the frequency of the host with the -s option.
```

1.2 Hardware Evaluation : autopcl tool

Timpol relies on the processor timer accuracy to report precise performance evaluation. If the update frequency of this timer is low, the loops with low execution times can-

not be accurately evaluated. Fortunately, nowadays processors offer special registers for dynamically evaluating the performance events at the hardware level. These registers (performance counters) can report different sorts of metrics, depending on the architecture (number of executed cycles, number of cache misses in the different cache levels, TLB misses, IPC, MFLOP, etc.). The hardware registers are accessed and initialized through some specialized OS kernel calls.

Tool Description

We have developed a tool called *autopcl* which uses the hardware performance counters of the target processor to accurately report the metrics of the performance of a code portion (typically a loop, but can either be any other code fragment). Actually, autopcl supports two platforms: linux/intel-pentium and solaris/ultrasparc-II. Autopcl instruments a fortran program to insert procedural calls to the PCL library [1]. PCL is a set of portable high level routines accessing the hardware performance counters of the target processor. The list of the events supported by autopcl are:

1. time events: number of elapsed clock ticks;
2. memory hierarchy events: number of cache misses and hits, with differentiating the read from the write access to the two cache levels, and for the two cache types (instructions and data);
3. TLB events: number of misses and hits for both the TLB of the data and/or instructions;
4. types of operations effectively committed: number of memory read and/or write, number of floating point operations, number of branches, total number of committed operations;
5. speculation: number of branches correctly predicted and/or miss-predicted;
6. multiprocessor systems: number of successful and unsuccessful atomic operations (test-and-set);
7. bottleneck: total number of stalled cycles, and those resulted from loads and/or stores and/or float operations and/or integer operations;
8. performance: MFLOP, IPC, miss-rate for the two levels of caches, machine balance (memory/FP).

For all the supported events, autopcl enables to count them during the two execution mode: user or/and system mode. Unfortunately, some of these events are not supported by the target processor. Autopcl can determine which events are supported. As example, the events supported by the ultra-SPARC-II are shown by the following autopcl output:

```
autopcl: instrument code sections to use performance hardware counters.
author : TOUATI Sid Ahmed Ali (INRIA 1999)
processor running at :  333 MHz

testing start/stop for available events:
----------------------------------------
```

```
PCL_L1CACHE_READ         : not supported
PCL_L1CACHE_WRITE        : not supported
PCL_L1CACHE_READWRITE    : not supported
PCL_L1CACHE_HIT          : not supported
PCL_L1CACHE_MISS         : not supported
PCL_L1DCACHE_READ        :        160394
PCL_L1DCACHE_WRITE       :         80084
PCL_L1DCACHE_READWRITE   : not supported
PCL_L1DCACHE_HIT         : not supported
PCL_L1DCACHE_MISS        :         41600
PCL_L1ICACHE_READ        : not supported
PCL_L1ICACHE_WRITE       : not supported
PCL_L1ICACHE_READWRITE   :        183838
PCL_L1ICACHE_HIT         :        175010
PCL_L1ICACHE_MISS        :            18
PCL_L2CACHE_READ         : not supported
PCL_L2CACHE_WRITE        : not supported
PCL_L2CACHE_READWRITE    :         80426
PCL_L2CACHE_HIT          :         90126
PCL_L2CACHE_MISS         :           271
PCL_L2DCACHE_READ        : not supported
PCL_L2DCACHE_WRITE       : not supported
PCL_L2DCACHE_READWRITE   : not supported
PCL_L2DCACHE_HIT         : not supported
PCL_L2DCACHE_MISS        : not supported
PCL_L2ICACHE_READ        : not supported
PCL_L2ICACHE_WRITE       : not supported
PCL_L2ICACHE_READWRITE   : not supported
PCL_L2ICACHE_HIT         : not supported
PCL_L2ICACHE_MISS        : not supported
PCL_TLB_HIT              : not supported
PCL_TLB_MISS             : not supported
PCL_ITLB_HIT             : not supported
PCL_ITLB_MISS            : not supported
PCL_DTLB_HIT             : not supported
PCL_DTLB_MISS            : not supported
PCL_CYCLES               :        894345
PCL_ELAPSED_CYCLES       :        754934
PCL_INTEGER_INSTR        : not supported
PCL_FP_INSTR             : not supported
PCL_LOAD_INSTR           : not supported
PCL_STORE_INSTR          : not supported
PCL_LOADSTORE_INSTR      : not supported
PCL_INSTR                :        523477
PCL_JUMP_SUCCESS         : not supported
PCL_JUMP_UNSUCCESS       : not supported
PCL_JUMP                 : not supported
PCL_ATOMIC_SUCCESS       : not supported
PCL_ATOMIC_UNSUCCESS     : not supported
PCL_ATOMIC               : not supported
PCL_STALL_INTEGER        : not supported
PCL_STALL_FP             : not supported
PCL_STALL_JUMP           : not supported
PCL_STALL_LOAD           : not supported
PCL_STALL_STORE          : not supported
PCL_STALL                : not supported
PCL_MFLOPS               : not supported
```

```
PCL_IPC                  :      0.798678
PCL_L1DCACHE_MISSRATE    : not supported
PCL_L2DCACHE_MISSRATE    :      0.007670
PCL_MEM_FP_RATIO         : not supported
```

An example of an instrumented file for determining the number of elapsed cycles in the innermost I-loop is as follows:

```
c -------------------------------------------------------------
c this file is automaticly generated by instpcl v 1.0
c          Touati Sid Ahmed Ali (INRIA, 1999)
c -------------------------------------------------------------
...
c pcl declaration section
        INCLUDE 'pclh.f'
        INTEGER PCLquery, PCLstart, PCLstop
        EXTERNAL PCLquery,PCLstart, PCLstop
        INTEGER counter_list(1), flags, res
        INTEGER*8 i_result
        REAL*8 fp_result

c pcl init section
        flags = PCL_MODE_USER
        counter_list(1) = PCL_ELAPSED_CYCLES
...
      do  k = 1,n
         do  j = 1,n

c pcl begins counting
        res = PCLquery(counter_list, 1, flags)
        IF(res .EQ. PCL_SUCCESS) THEN
           IF(PCLstart(counter_list, 1, flags) .NE. PCL_SUCCESS) THEN
              WRITE(*,*) 'problem with starting event ', PCL_ELAPSED_CYCLES
           ENDIF

c Innermost loop
           do  i = 1,n
            fortran statements of the loop
           enddo

c end couting and reporting results
           res = PCLstop(i_result, fp_result, 1)
           IF(res .NE. PCL_SUCCESS) THEN
              WRITE(*,*) 'problem with stopping event  PCL_ELAPSED_CYCLES '
           else
              WRITE(*,*) 'PCL_ELAPSED_CYCLES: ', i_result
           endif
        ELSE
         WRITE(*,*) 'event PCL_ELAPSED_CYCLES not supported by your processor'
         STOP
       ENDIF
...
```

Autopcl executes the instrumented program n times ($n = 10$ by default), compiled with an optimization level specified by the user. It reports statistics on the achieved dynamic events (min, max, mean, variance) and store them in a database: if the performance has high variations from one execution to another, the code fragment should be OS-sensitive. An example of autopcl output is as follows:

```
event                Min           Max        Mean          Var
PCL_ELAPSED_CYCLES   158684502     228322584  169807674.44  491388499287382
PCL_L1DCACHE_MISS    898032        906838     901317.33333  9807810
PCL_L2CACHE_MISS     52347         57982      54510.555556  3100271.7777
PCL_IPC              0.9313432     0.9387215  0.9361295052  7.211900e-06
```

Usage

Autopcl is called as follows:

```
ariane:~> autopcl
          TOUATI Sid Ahmed Ali (INRIA 1999)
autopcl: instrument code sections to use performance hardware counters.
usage: autopcl -i <inputfile>.f [-o<outputfile>.f] -p <pcl_event> [-x|-c] [-r]
                  [-f compile_option] [code section delimitation] [-m <pcl_mode>]
                  [-d <data_input_file>] [-l] [-t] [-n number]

 results are gathered in the mini database <inputfile>_autopcl.stat
 input-file: file.f without .f, fortran file to analyze
 unparsed_file: file.f to generate (to stdout by default)
 pcl_event: the event to be catched by the performance hardware counter
 code section delimitation (if omitted, analyze all the program)
    you could use line numbers, like this :
        =b num_begin_line
        =e num_end_line (optional if b points to a loop)
    or you could use markers, like this :
        -[ string marking the beginning of the section
        -] string marking the ending of the section
 options:
        -x for compiling the fortran generated file and execute.
           Results are in autopcl_results.
        -c for only generate an executable file
        -r for remove binary generated files.
 -f compiler optimization.
 -l list all pcl_events
 -t to test events that are supported by your processor
 -m pcl_mode:  PCL_MODE_USER, PCL_MODE_SYSTEM, PCL_MODE_USER_SYSTEM
 -d data_input_file: the file to be given to the program as input data
 -n number of execution. 10 by default
```

2 On the Prediction of Loop Performance Optimality

This section studies the problem of the optimality computation for the innermost loops. We would like to answer the question "what is the best performance that an innermost loop could reach?". The answer has a particular importance because:

1. if the achieved performance is equal (or close) to the optimal one, the considered loop doesn't have to be optimized;

2. if not, the optimal performance becomes a target (a goal) for the optimization process.

The best performance of a loop is determined by complex and correlated factors, mainly we cite: the fine grain schedule, the cache effects, branches and OS interactions. Even

if we only consider the schedule under resource constraints, looking for the best schedule time is NP-complete. This section does not study the optimality in the strong sense, but with a released definition closer to "optimistic performance" than to "best performance".

To help the end-user for understanding the source of the bottlenecks of his loops, we must report the contribution of the different factors to the performance limits. We investigate the three main factors: the data dependences which define the intrinsic ILP available in the loop, the resource constraints which describe the processor functional units limitations, and a combination between the while including the cache effects.

The two first factors (data dependences and resource constraints) can be figured out statically as we will see in the next section.

2.1 Static Optimality

The optimal performance of a loop is first constrained by the inherent fine grain parallelism defined by the code. The end-user must write a program which exhibits sufficient amount of ILP. The data dependence relations between the statements are computed statically at the high level code (fortran program source) in order to be as close as possible to the programmer in an interactive way: a programmer usually understands better his code than the generated assembly instructions. The following section describes how we provide the programmer with a framework for computing the optimal performance of a fortran innermost loop. This information should help him to tune and rewrite his application to get the lowest possible performance limit.

Intrinsic ILP : visudep tool

Visudep is a interactive tool which takes as input a fortran loop and provides the following information to the user:

1. the data dependences between the operations;
2. the performance limits of the loop defined by the data dependences;
3. the dependences which defines these limits.

The fortran source is parsed using the SAGE++ framework [4] and the data flow analysis is done using the omega test [6]. Our tool reports the data dependence information in two manners: either in a textual way, or with an interactive user interface using xvcg [5]. In the latter case, the data dependence graph appears where its arcs are labelled by the distance vectors with different colors depending on the type of the dependence (flow, anti, output). The user can chose to focus on a subset of the dependence type, for instance on flow dependences. The node of the data dependence graph can be either the source line instructions (each fortran instruction is considered as atomic), or the three address operations (the fortran instructions are decomposed into three address code by visudep).

After building the data dependence graph, our tool computes the critical circuit and reports it to the user, which is the minimum execution rate of a software pipelining schedule. The end-user can tune his loop in order to reduce this critical circuit which constitutes

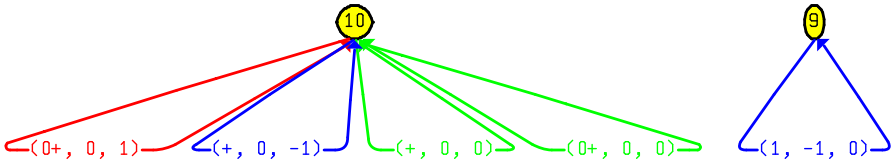


Figure 4: Data Dependence Graph of the Fortran Statements

the limit of the ILP.

As example, consider the following fortran loop :

```
do k=1,n
   do j=1, n
      do i=1,n
         A(i,j,k) = x* A(i,j-1, k+1) * B(i-3,j+1, k)
         C(i, j)=C(i-1, j) / B(i,j, k-2)
      enddo
   enddo
enddo
```

Visudep reports the data dependences of the I-loop textually as:

```
------> ANTI dependence between c(i - 1,j) (line 10) and c(i,j) (line 10)
        with vector (+, 0, -1)
------> FLOW dependence between c(i,j) (line 10) and c(i - 1,j) (line 10)
        with vector (0+, 0, 1)
------> OUTPUT dependence between c(i,j) (line 10) and c(i,j) (line 10)
        with vector (0+, 0, 0)
------> OUTPUT dependence between c(i,j) (line 10) and c(i,j) (line 10)
        with vector (+, 0, 0)
------> ANTI dependence between a(i,j - 1,k + 1) (line 9) and a(i,j,k) (line 9)
        with vector (1, -1, 0)
```

Graphically, the data dependence graph represents either the fortran statements (see the output of visudep in Fig. 4, the nodes are labelled by the source line number) or three address code (see the output of visudep in Fig. 5). The arcs are labelled by the distance vectors (the non labelled arcs represent the non recurrent dependences). The flow dependences are represented by red arcs, anti dependences by blue ones, and output dependences are in green[3]. The shape of the nodes of the three address code indicates if it is a load (rectangular shape) or a store (triangle)[4], or an arithmetic operation (circle). The optimal performance defined by the data dependences on the three address code is reported by visudep as:

```
critical circuit: 16/1
```

The critical ratio 16/1 means that the optimal performance is 16 cycles per iterations. The dependences which contribute to this limit are reported as:

[3]We apologize for the readers who have a black/white version of this report.

[4]We assume that scalars are in registers and hence does not require a load nor a store.

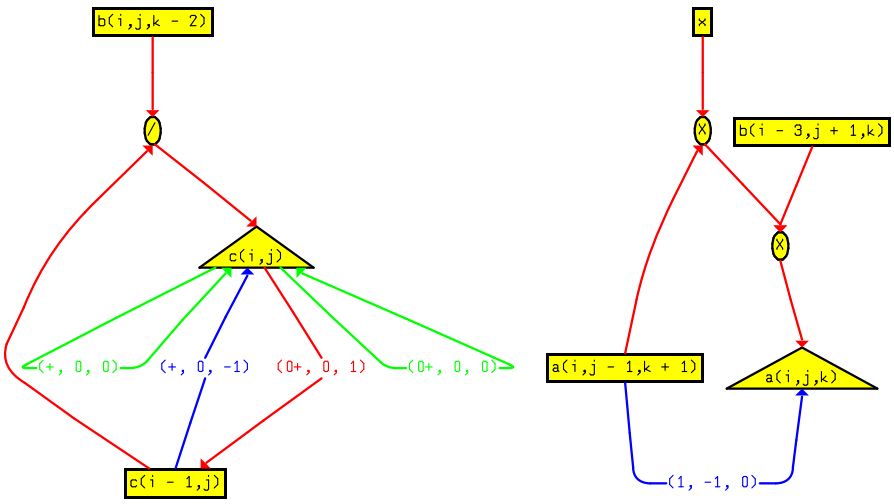


Figure 5: Data Dependence Graph of the Three-address Code

```
source:  expression: c(i - 1,j) at statement at line 10.
dest:  expression: c(i - 1,j) / b(i,j,k - 2) at statement at line 10.
<2 , 0>

source:  expression: c(i - 1,j) / b(i,j,k - 2) at statement at line 10.
dest:  expression: c(i,j) at statement at line 10.
<12 , 0>

source:  expression: c(i,j) at statement at line 10.
dest:  expression: c(i - 1,j) at statement at line 10.
<2 , 1>
```

A couple $< x, y >$ defines the latency of the operation and the distance carried by the innermost loop. Note that the latency of the operations are assumed to be those of the ultra-SPARC processor by default.
An important remark is that visudep computes the limits of the ILP of a loop as it is, i.e. the tool does not perform any high or intermediate level code optimization (redundant memory operations and peephole optimization, common sub expressions eliminations, etc.). These optimizations must be done before using visudep (by hand, or automatically using ictineo or Sage++) in order to get realistic performance limit.

Visudep is called as follows:

```
ariane:~> visudep

         TOUATI Sid Ahmed Ali (INRIA 1999)
visudep: do data dependence of a loop in a program and generate a vcg file format.
usage: visudep -i <inputfile>.f -l loop_line [-v] [-d]
     -i: input file to analyze (without .f)
```

```
-l: line number of the loop to analyze
-d: decompose the fortran statements (3-address). Data dependences are displayed
    between expressions rather than fortran statements.
-v: visualize data dependence interactively.
```

Processor Limitations : MIIress tool

Using the high level code in visudep to interact with the end-user is a flexible and convenient approach. However, the source code may be far away from the real program executed by the processor, because the processor can have some characteristics which make the compiler generating low level code with new operations (address calculation, tests, special instructions, static speculations, etc.). If the end-user wants to optimize his loop for a target processor, he must be involved within the low level generated code in order to understand the limitations of the target machine.

For this purpose, we developed *MIIress* using the Salto framework [2] to report the optimal performance regarding the generated assembly loop on a specific architecture. MIIress takes as input the low level code of a loop body, a processor description (functional unit descriptions) and an instruction set. The reservation tables of the instructions enable us to statically compute the number of cycles spend in each FU during the execution of a loop iteration. MIIress analyzes the assembly code and report the limit of performance caused by the FU constraints, and which FU constitutes a bottleneck. Actually, MIIress supports the ultra-SPARC and SPARC architectures under the V9 instruction set.

When using the compiler for generating the assembly code, the user must turn on the compiler optimizations in order to eliminate the redundant operations. As example, the output of MIIress for the low level loop defined in the last section and generated by the Sun compiler without any optimization is:

```
Total cycles spend at each FU type ...
Resource        #Copies          Cycles/it
issue            4                158
issue_flp        2                3
alu              1                122
ldst             1                32
branchunit       1                1
flp_add          1                2
flp_div          1                12

Minimum II is constrained by ...
alu : 122 cycles/it
```

However, using the compiler optimization level -O1 (peephole optimization), the output becomes:

```
Total cycles spend at each resource type ...
Resource        #Copies          Cycles/it
issue            4                116
issue_flp        2                3
alu              1                86
ldst             1                26
branchunit       1                1
flp_add          1                2
flp_div          1                12
```

```
Minimum II is constrained by ...
alu : 86 cycles/it
```

Depending on the compiler, the user must chose the optimization level which produces the lowest performance limit. The best optimization level that we suggest on ultra-SPARC must at least be -O2 (up to -O5):

```
Total cycles spend at each resource type ...
Resource       #Copies         Cycles/it
issue           4               17
issue_flp       2               3
alu             1               7
ldst            1               6
branchunit      1               1
flp_add         1               2
flp_div         1               12

Minimum II is constrained by ...
flp_div : 12 cycles/it
```

For our loop, the minimum execution rate of the loop is then 12 cycles because of the floating point division FU. Using visudep on the high level and MIIress on the low level, the end-user can know if the loop performance is limited by the processor itself or by its code (data dependence).

MIIress is called as follows:

```
Usage: MII_ress -i <asmfile> -m <machine description> [options]
Available options are:
  -v, --version          shows version number
  -s, --silent           silent mode
  -W, --warning          display warning and debug messages
  -c, --inline           don't leave "call" alone in a basic block
  -l, --language         high level language used (actually, it is always fortran !)
  -m, --machine          path to the machine description file
  -i, --input            input assembly file
  -h, --help             display this help message
      --credits          display the list of contributors
  --                     stops processing args
```

Finally, we note that it is up to the user to give the correct assembly code section related to the high level program. MIIress does not automatically build a correlation between the high level and the low level code. The end-user must map between the two levels.

2.2 Dynamic Optimality : instop tool

The static analysis on a loop enables us to get a quick feedback regarding the optimality. In the presence of branches inside the loop body, we cannot determine at compile-time which direction the control flow would take during the execution since in most of time it depends on the input data set and on the dynamically computed variables. Our static tools (visudep and MIIress) are conservative and assume that all branches are taken: this yields to over estimating the performance optimality since non taken branches does not contribute to the execution time. Furthermore, the static tools does not report the interaction between the data dependence constraints and the resource constraints.

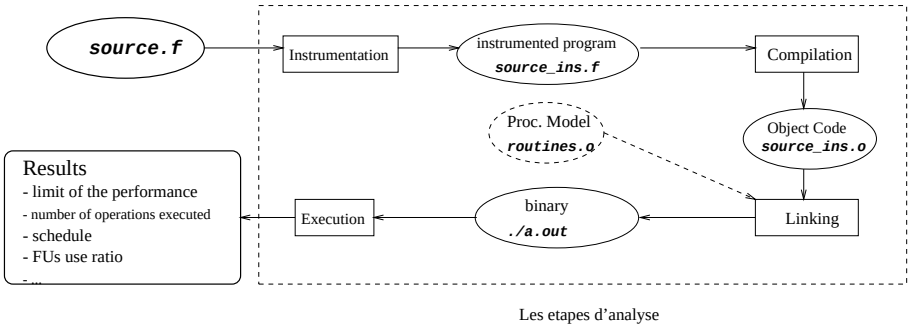


Figure 6: Instop Steps

Tool Description

We have developed *instop* to compute the optimistic performance of a loop by considering both the data dependences and the processor characteristics. Our tool takes as input a high level source code and a processor model. It executes the code and catches on-the-fly the operations (three address code) effectively executed. It builds dynamically a new optimistic schedule for these operations by using a list scheduling heuristic. Our tool builds an as soon as possible schedule of the completely unrolled loop. This is equivalent to assume an infinite issue buffer size and ROB (ReOrder Buffer) size of the target processor in which the loop executes. The constructed schedule constitutes the best performance that we can sustain from the loop on the considered processor.

Figure 6 shows how instop works. The first step decomposes the code into three address operations and instruments it by inserting procedural calls to dynamic routines. These routines are responsible for catching on-the-fly the dynamically executed operations and schedule them in respect of the processor characteristics. The instrumented program is compiled and linked to our routines. The execution of the final binary reports the optimistic schedule, the FU use ratio and the number of effectively executed operations. The user must give a representative input data set to the instrumented program in order to get realistic results.

The scheduling process is done by the instrumentation routines as noted above. Actually, instop supports an ultra-SPARC processor model. The routines catch each new executed operation and schedule it in the first free slot of the needed FU after resolving the data dependences. For these latter, we keep for each data element (scalars and array elements) used in the loop the date when it would be available, see Fig. 7. When an operation is executed in the loop, the instrumentation routines look for its operands and their availability time. The operation must be scheduled after the availability date of its operands. When a free slot is found, the operation is scheduled and the availability time of its result is updated.

To take into account the cache effects, the latency of a load operation must be adjusted depending if it is a cache miss or not. Actually, instop supports two memory hierarchy

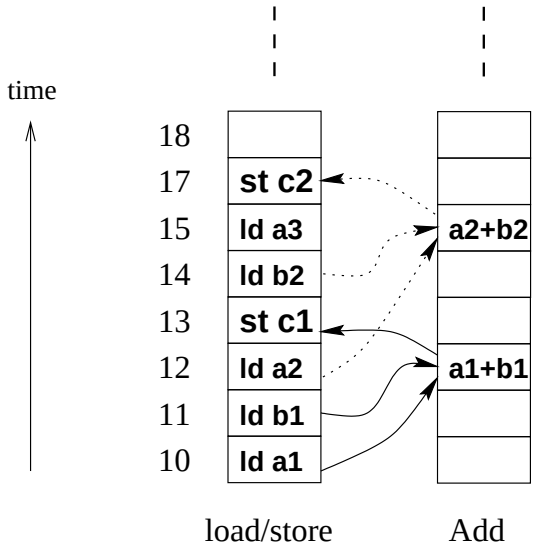


Figure 7: Dynamic Constructed Schedule

models:

1. optimistic model, where all the memory accesses are assumed to be hits in the cache;
2. compulsory model, where the first load which accesses a cache line is a miss and the other successive loads to this cache line are hits. Instop handles a bit for each cache line, set to 1 if the line is accessed. Note that only array access need load operations, and we assume that scalars reside in registers.

An example of the instop output is as follows:

```
Optimal number  of cycles              : 4915
Optimal time in seconds               : 2.94900E-05
Optimal cycle per itération             : 4

Number of opérations for each resource type
 Ld_St  Iadd   IaddMult  Fadd  FMult  FDiv
 4800   0      0         1200  1200   0

ultraSPARC use ratio :
 Ld_St  Iadd   IaddMult  Fadd  FMult  FDiv
 0.98   0.00   0.00      0.24  0.24   0.00

scheduling trace:
......
Cycle  Ld/St FpAdd FpMul OPERATION
153    X     X     X     ld A(2,18), A(2,7)*B(7), [A(2,3)*B(2)]+C(80)
154    X                 ld B(19)
```

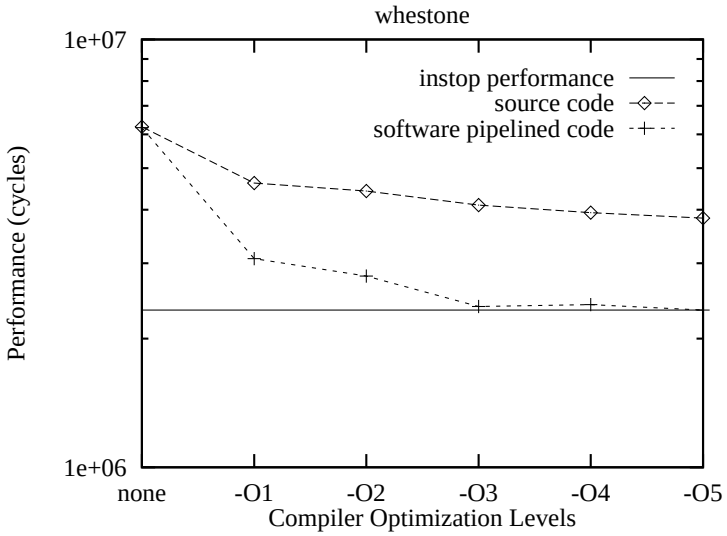


Figure 8: Achieved Performance vs Performance Limit for Whetstone

```
155    X     X           ld A(2,19), [A(1,17)*B(1)]+C(80)
......
```

Figure 8 is an example of our experimentation done on ultra-SPARC for a fortran loop taken from the whetstone benchmark. This loop had the particularity of mainly containing computations on scalars. The horizontal line in the figure is the optimistic performance reported by instop with the compulsory miss model. The Sun compiler did not succeed in achieving this performance limit for the original code, even with the highest optimization level (software pipelining is done at level -O4). We used the *tops* tool [3] in order to build a software pipelined loop at the source level. We managed to achieve this limit, and hence we could stop the optimization process. Since the loop did most of its computations on scalars, the cache effects were not a source of bottleneck, and hence we achieved the best performance.

Unfortunately, not all the loops could reach the optimal performance. Figure 9 shows the results of the performance optimization process of the matrix multiply kernel. Even with high level optimization techniques (tiling, software pipelining, loop permutation), the achieved performance remained far from its limit. The main reason is that the ultra-SPARC does not recover dynamically the loads which missed the cache, and the Sun compiler does not schedule the low level operations according to this fact. In this case, tuning the loop in the low level is a better approach than optimizing the source of the application.

Usage

Instop is called as follows:

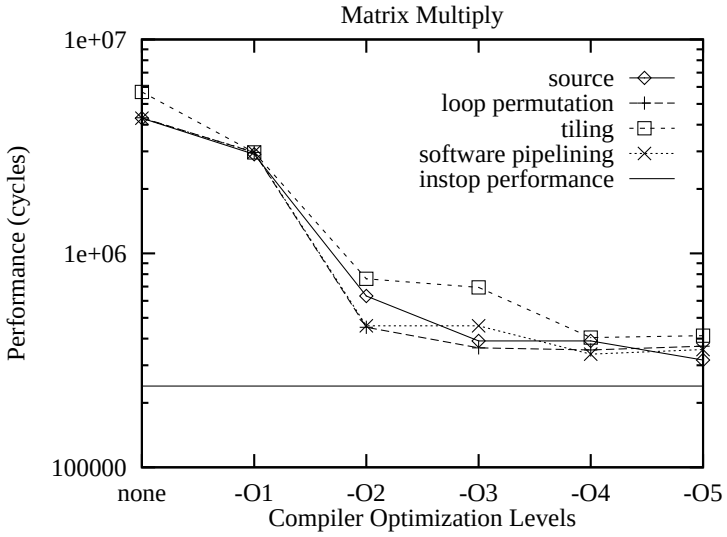


Figure 9: Achieved Performance vs Performance Limit for Matrix Multiply

```
ariane:~> instop
         TOUATI Sid Ahmed Ali (INRIA 1998)
instop: instrument fortran programs for optimistic performance evaluation
usage: instop -i <inputfile>.f [-o<outputfile>.f] [himem_options]
              [code section delimitation][-x][-v][-n][-f][-h]

 input-file: file.f without .f, fortran file to analyze
 unparsed_file: instrumented file (to stdout by default)
 code section delimitation (if omitted, analyze all the program)
        =b num_begin_line
        =e num_end_line (optional if b points to a loop)
 options:
        -x for compiling the fortran generated file, link and execute.
           Results are in instop_results.
        -v for verbose mode. Keep trace of computed operation schedule.
 himem_options: [-u, -c, -p, -d] (compulsory by default)
        -u use uniform memory access time
        -c compulsory miss
        -p probabilistic model (not available yet !)
        -d ponderated probabilistic model (not available yet !)
 -n : use nohup to execute.
 -f : use only data flow constraints to schedule operations.
 -h : do not use a memory hierarchy model.
```

References

[1] Rudolf Berrendorf and Heinz Ziegler. PCL – the Performance Counter Library. Electronic Document, Central Institute for Applied Mathematics at the Research Centre Juelich , Germany, July 1999.
`http://www.fz-juelich.de/zam/PCL/`

[2] F. Bodin, E. Rohou, Z. Chamski, and A. Seznec. The SALTO project. Electronic Document, IRISA-Rennes, 1997.
`http://www.irisa.fr/caps/PROJECTS/Salto`

[3] Min Dai. *Transformation et Optimisation des Programmes Source pour le Pipeline Logiciel.* PhD thesis, INRIA-Roquencourt, Université de Versailles, June 2000.
`http://www-rocq.inria.fr/~dai/Publications/these.ps.gz`

[4] Dennis Gannon and al. PC++/SAGE++. Indiana University, University of Oregon, Université de Rennes, January 1999.
`http://www.extreme.indiana.edu/sage`

[5] I. Lemke and G. Sander. Visualization of Compiler Graphs. Technical Report Design report D 3.12.1-1, Universität des Saarlandes, 1993. ESPRIT Project 5399 Compare.
`http://www.cs.uni-sb.de/RW/users/sander/html/gsvcg1.html`

[6] William Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8):102–114, August 1992.
`http://www.cs.umd.edu/projects/omega/omega.html`

Part IV
Mapping between source and assembler with ATAC

Abhishek Prabhat (INRIA, France, and IIT Delhi, India)

1 Motivations

1.1 Brief introduction to SALTO

SALTO stands for "System for Assembly Language Transformation and Optimization". This tool models the assembly code as objects to provide object based user interface which allows to easily implement complex transformation directly on the abstract representation of assembly programs. The user can write his own tool and experiment with the program, collect various statistics, and perform different alterations to optimize his code.

Essentially when given the assembly code (along with machine description and other parameters as described by SALTO) the tool builds a structure of the program and presents the user with objects like CFG, BB, INST etc. The user can now manipulate them to achieve its end.

1.2 Need for ATAC

ATAC is yet another tool made to utilize SALTO more efficiently and conveniently. This tool allows many users to work simultaneously on the same code. It comprises a server which owns the code and which actually does the processing on the code, and there are one or more clients which repeatatively load the necessary part of the code from the server, and inform of their query and changes to the server as and when required. Needless to say that mutual exclusion, and consistency is maintained by the server. The client can get exclusive rights on part of the code, work out, and then submit the changes to be visible to the world (something like the cache coherence protocols).

1.3 Objectives

The objective of this document is to explain the extensions performed in the provided ATAC version (29/5/2000). The tool was extended to provide the user with a set of menus, enabling him to view and comprehend the relation between the source code and the corresponding assembly code. The work had been pursued under the impression that given the exact knowledge of what lines in the assembly code correspond to which lines in the source code, it would be a great aid to the end user to help him better understand his code and implement his optimizing altercations.

2 Groups and mappings

2.1 What is a mapping?

A compiler when given a correct HLL program outputs the appropriate LLL program. We know therefore that the assembly code lines correspond to the source code. In our present context however we need more refined info. Or to be more specific we require the complete knowledge of what lines in the assembly code corresponds to which lines in the source code. It is this knowledge that is refered to as 'mapping' throughout the text.

2.2 What is a Group?

A 'Group' is an object which represents a part of this one to many mapping. A Group consists of the line number in the source file, and a link list of instructions that corresponds to the high level instruction at the given line number in the source file.

Server side:

```
class Group {
        int sourceLine;
        INS_B *head;
        Group *next;
}
```

'head' is an object of type INStruction Block, which for all practical purposes can be assumed identical to Instruction. The class was invented to find an easy representation of a link list of instructions. This class essentially contains only two fields.

```
class INS_B {
        INST *inst;
        INS_B *next;
}
```

The following methods are provided by these classes:

`Group()` : constructor. Sets sourceLine=-1 and head = null

`Group(int)` : constructor. Sets sourceLine as desired and head = null

`Group(int, INS_B)`: constructor. Sets sourceLine and head as argument.

`int getSourceLine()`: returns the sourceLine of the current Group.

`void setSourceLine(int)`: Sets argument as the sourceLine.

`void addInstruction(INST *)`: makes an Instruction Block of the specified instruction and adds it to the end of the list repersented by head.

`Group *getNextGroup()` :	returns the Group next to the current one. Returns null if no such Group.
`void setNextGroup(Group *)` :	sets the next Group of the current Group as specified
`INS_B *getHead()` :	returns the head of the link list of Instruction block. Returns null if the list is empty.
`void produceCode(FILE *)` :	writes the sourceLine followed by the list of unparsed instructions in the Group to the File passed in the argument.
`char *toString()` :	returns a string which contains the textual representation of the Group. The representation of the Group is as follows: (<sourceLine> (<textual representation of head>)(..)...(..)) This string is added as an attribute of type '10' (an arbitrarily chosen positive number) since the user-made attributes must to have a positive type. However the attribute information is not passed in the present ATAC and hence renders this function useless.

Client side: (note that the client is implemented in java)

```
class Group{
        int sourceLine;
        INS_B head;
        Group next;
}
```

The methods provided by this class have the same name and functionality as on the server side.

```
Group()           : constructor
Group(int, INS_B) : constructor
int getSourceLine()
INS_B getHead()
Group getNext()
void setNext(Group)
void produceCode()
```

Instruction Blocks are implemented slightly differently than the one on server side. The difference is essentially due to the difference in the representation of the Instruction. In `INS_B` the instructions are repersented as 3-tuple. (cfg, bb, inst). These three integers represent the position of instruction in the procedure (cfg), basic Block (bb) and the position within the basic block (inst).

Methods `INS_B(int, int, int)` : constructor.
Sets the cfg, bb, inst respectively. Writes error message if -ve arguments and sets the instruction as (-1, -1, -1) which is indicative of invalid instruction.

`void setNext(INS_B)` :

`INS_B getNext()`

`int get_cfg()` : returns the position of cfg in which the current instruction is situated.

`int get_bb()` : returns the position of the BB within the CFG in which the instruction is situated.

`int get_inst()` : returns the position within the BB where the instruction is situated.

3 How are Groups made?

The crux of this project has been the formation of Groups. Given the assembly code and the source code how can one figure out the mapping between these two ? Apart from this everything else is an implementation issue. In the present course of action we have to completely rely on the compiler for this.

With the GNU compiler for C gcc this information can be extracted using the debug option -g. This option puts in the assembly code, assembler directives such as .stabs and .stabn which contain the various details about the code, including the line number information. We focus on the .stabn directive and try to learn its format.

```
.stabn type, other, desc, value
```

Type-information is either a type-number, or 'type-number='. A type-number alone is a type reference, referring directly to a type that has already been defined. A type-number= 68 corresponds to the information that we seek. Notice that the stabn directives with a type value other than 68 does not contain the mapping info. The exact interpretation of the fields 'other', 'desc' and 'value' depends on this type value.

In our case the 'other' field is invariably set to zero.

'desc' contains the line number in the source file which is the root of the later described set of lines.

'value' contains Local label which maps to the source line. Local labels are the series of labels starting with 'L'. These labels are incorporated by the compiler for debuggers and are not meant for the users. Recall that the stabs, stabn directives are also incorporated for the debuggers.

For instance, the instruction '`.stabn 68,0,20,.LM6-main`' suggests that all the instructions following the Label `.LM6` (before upto the next `.LM` label which would be `.LM6`) would be mapped to the 20 th line in the source file. Additionally the value field '`.LM6-main`' also suggests that these instructions are a part of function 'main'.

Each stabn is generally followed by the corresponding LM label (as described in its value field). So one could have easily used the .stabn directives as the boundary of the Group. However such a mapping would have depended on the exact position of these stabn directives and not the LM labels. Rather we have relied on the information provided for the LM labels and then find the Group boundaries using these labels. One advantage of this scheme over the previous one is that even if the stabn instructions are shuffled (leaving the labels intact) the mapping would not alter. The inclusion of .stabn instruction is not a issue as these are not processor instructions. In fact SALTO categorizes them as Pseudo Instructions.

3.1 A Pitfall

SALTO provides the user with Objects such as CFG (procedures), which consist of many basic blocks BB. These basic blocks in turn consist of many instructions INST. Ideally enumerating these instructions in each BB, and enumerating each BB and so on recursively, one should reproduce the assembly code (except perhaps some minor replacement for some particular type of instruction depending on the machine description). However with the Pseudo Instructions '`.stabn....`' a strange phenomenon was observed. In the course of implementation it was revealed that some (not all) of these .stabn lines were missing from the enumeration. Because of these missing lines the Groups formed were bigger than otherwise and of course faulty.

These missing stabn lines were not attached to any BB but rather fell between two BB. Consider the following piece of code :

```
        mov %i1,%o1
        ld [%l1+%lo(max)],%o2
        cmp %o0,%o2
        bge .LL3                -> Branch instruction.
                                   BB ends at the next line
        sll %i1,1,%o1           -> BB ends here because of delayed branch

.stabn 68,0,19,.LM5-main        -> Pseudo Instruction, hence BB not begins

.LM5:                           -> beginning of next basic block
        srl %o2,31,%o0
        add %o2,%o0,%o0
        sra %o0,1,%o0
        cmp %o1,%o0
```

As you see, being a Pseudo Instruction it was left out strangely some times. This precisely explains the faulty behaviour of the assembly $->$ source mapping of the previously existing ATAC version. Still another drawback with the previously implemented one way mapping was that it maps each BB to a source line. However basic block not being the atomic entity might map to more than one source code lines. Highlighting a single line might mislead the end user in such cases.

The problem still remains as to how to recover the missing stabn instructions. SALTO however provides the answer to the same. Apart from the hierarchy of CFG, BB and

INST, SALTO also provides the representation of the code as a flat list of Instructions (INST). No Pseudo instructions are missing in this list. Hence the information is picked up from the stabn appearing in the flat list and the Groups were made using the LM labels and instructions in the hierarchy.

The formation of Groups was done in the function `Salto_hook()` in the file `salto_server.cc`. The link list of Groups thus formed can be accessed through the variable '`first_Group`' in the same file.

3.2 Transfer of Groups

In order to transfer this Group information to the client side, the first scheme adopted was to attach this Group as an attribute to each instruction, and with the usuall transfer of instruction this would be available on the client side. SALTO has facility of attaching user made attributes to the instructions. Thus attributes were made of these Groups and hooked to the respective instructions. However it was later realized that only predefined attributes (and perhaps only COMMENT_ATT) is passed along the instruction. The problem was resolved in a different way. The client when establishing connection with the server sends a request to transfer the group info. The server then sends the group information following a simple protocol. Simultaneously the client builds INS_B and Groups as and when the information arrives. Successful completion of the transfer protocol results in the formation of the image of the Group list on both sides. With this information available on both sides any manipulation and traversal can be done at will.

3.3 How to access an Instruction?

The group information available gives the representation of the instruction as a 3 tuple (cfg, bb, inst). The required CFG can be accessed through function `getCFG (int)` provided by ATACData. Or the appropriate BB can be accessed through the function `getBB (int cfg, int BB)` of the same. Ideally the desired instruction should have been possible to obtain through function `getINST(int inst)` of BB. But this function invariably returns null (which is because the BB on the client side is not further fragmented into Instructions) because of which a less efficient solution was adopted. The basic block consisting the instruction is first loaded (if not loaded) and then the desired instruction is selected from the assembly code table currently displayed on the client window.

3.4 Some singular points about the mapping

- The Pseudo instructions in the beginning of the code '.stabs' directives are not accounted in any group. In fact they are not part of any BB either, which explains why they are outcasted from the mapping. SALTO puts these instructions as prologue.
- The first few instructions in the first basic block, the ones that appear before LM1, they again do not form part of any group. But this is acceptable as these are 1-2 in number and are not processor instructions. Generally name of the procedure or so.
- The mapping obtained is not onto. There may be source code lines corresponding to which there is no Group in the list. This happens when the statement semantically

ends in the next line (such as declaration of a function) or sometimes when the statement has already begun on a previous one (such as the end of a loop).

- The list of Groups when traversed in increasing order generally yields code corresponding to increasing line number in the source file. However at times (end of loops or functions) the assembly code maps to a previous line number.

4 Features of the tool

4.1 Features added in the client window

The client window is divided into 3 parts. The leftmost frame depicts the possible selections in the hierarchy, the middle frame contains the table of the loaded module (which might be a CFG, or a BB, or a Group) and the right most contains the source code represented as a list. The menubar in the window has been extended to include a new menu 'Map'. This menu has the following 6 options:

1. Source Line

   - When you select an instruction in the Basic Block and activate source Line, the Group corresponding to that instruction is searched and the source Line of the rightmost frame is highlighted. Furthur it also loads the assembly instructions in the Group in the middle window. If no such Group exists pops up an error message saying 'no Group exists for your query'. This error may occur because of following reasons
   - The selected instruction was special marker instruction denoting Beginning or end of the BB. Do not select the blank instructions in the beginning or end of the BB.
   - The display mode is not in BB mode. Either a cfg is loaded in the window or a Group has been loaded. Select the appropriate BB from the hierarchy in the left and then select the desired instruction.

2. Assembly Code

   - When a line is selected in the source code and the user wants to view the corresponding source Code, he should activate Assembly Code menu item. This searches in the entire list of Groups the one with the selected line number, and then load the group. This function is uneffected by the Display mode. However if no such Group is found it pops up an error message saying 'Sorry! No group found for your query'. In such cases the user is advised to go for 'Next Nearest Group' or 'Previous Nearest Group', whichever he thinks appropriate.

3. Next Group

   - The client stores in the memory the last Group loaded. The next Group in the list is loaded at this request. In case it is the tail an error message is generated.

4. Previous Group

- Loads the Group previous to the one in memory. If memory is clear or if the previous group is null, generates error message.

5. Next Nearest Group

   - Loads the group with the source Line equall to or just greater than the selected line. Generates error message if there is any.

6. Previous Nearest Group

   - Loads the Group with the source line equall or just smaller than the selected line. Generates error messasge reporting the exact error, if there is any.

4.2 How to run?

1. SALTO [2] and ATAC [1] must be installed and their environment variables correctly set ;

2. set the java CLASSPATH environment variable and insert the path to the swing-1.1 library and the path to the java client ;

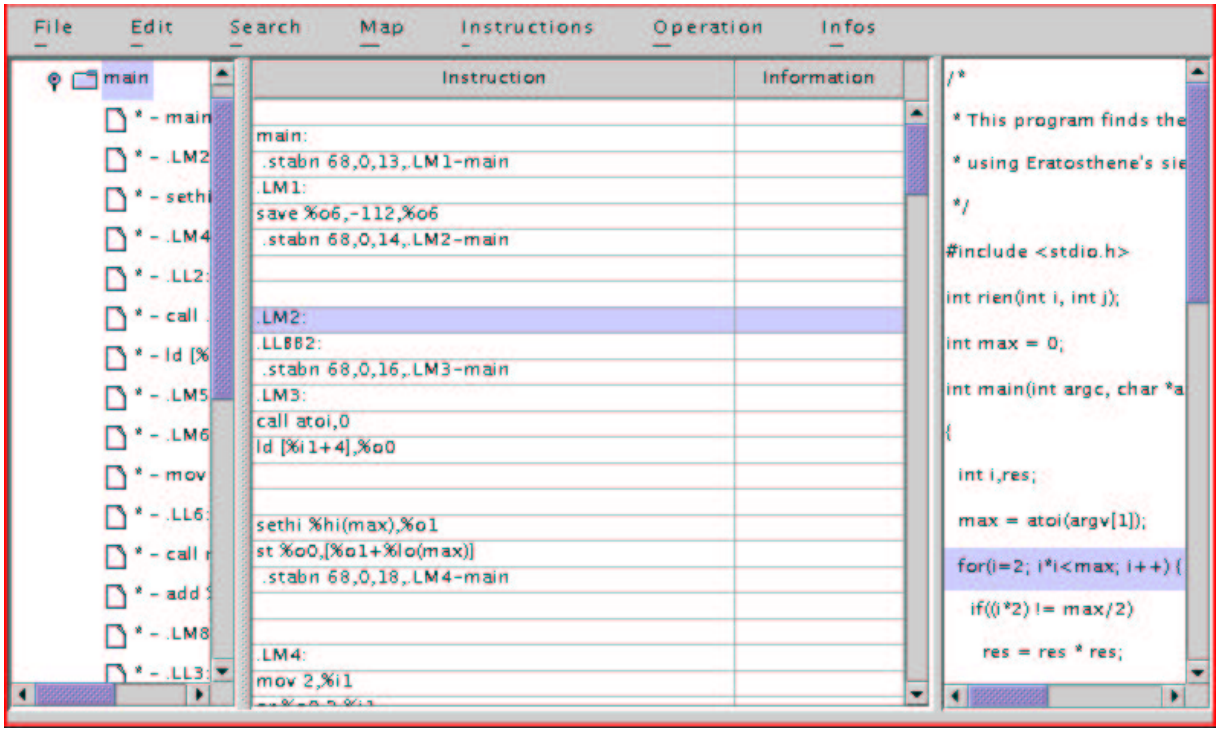
3. start the server on a machine by calling the script :

   ```
   ./go input_assembly.s
   ```

4. start the tool by calling the java application (java version 1.2) :

   ```
   java ATAC server_host input_assembly.s source_prog.c
   ```

Figure 10: Snapshot of modified ATAC window on an example.



5 Possible Improvements

- Presently the mapping is assumed to be one to many (`1->n`) i.e. one source code line corresponds to n assembly code lines. However this might not always be true. i.e. there may be a set of source code lines that corresponds to a set of assembly code lines. This 'many to many' mapping cannot be represented through such groups. However the present Group class can be easily extended to contain a link list of source lines (very similar to the list of instructions). This situation can arise in optimized code.

- Presently the Groups are made only through the stabn directives put by the GNU compiler. However this is the standard only for GNU and other compilers have not adopted this. The f77 compiler for fortran puts the line number information by putting comments in the file. The comments begin with '!' are of the form

```
!<line number> <HLL instruction>
<Assembly Instruction 1>
<Assembly Instruction 2>
```

```
.
.
.
<Assembly Instruction n>
```

All the instructions 1 to n map to source line number <line number> in the comment. These comments are included even without the debug option. However with optimization such a mapping might be lost. Moreover many of the debuggers do not work with optimized code. With the advent of class Group on the server the user can write a function `mapsource()` which reads the assembly code and decides the mapping, in case the stab directives are not available, and hook it in `Salto_hook()` on the server side. Rest of the things follow automatically. Thus the domain of this facility can be increased gradually to encompass more and more compilers.

- Still further, one could devise techniques to figure out the mapping without being provided any extra (debug) information by the compiler. A plausible scheme for the same could be to insert a peculiar HLL instruction in the beginning of the source code and record the low level instructions generated for it. Now this line can be inserted after the first and then note the assembly code for the same set of instructions. Thus recognizing the definite pattern produced by the dummy source line, and moving this dummy statement by one in each iteration, one would observe a similar motion in the observed pattern. Thus the differences in each iteration can reveal the mapping to some extent. This scheme however requires extensive overheads for compilation and may prove unpractical for long programs.

References

[1] Hervé Guihot, Thomas Hérault, Hervé Le Bihan, Alban Maillère. ATAC: Analysis Tool for Assembly Code Final Study Project, IFSIC, Rennes, France. February 2000.

[2] F. Bodin, E. Rohou, Z. Chamski, A. Seznec, The SALTO Project. IRISA-Rennes. 1997.
`http://www.irisa.fr/caps/PROJECTS/Salto`.

Part V
Debugging Optimized Code

Abhishek Prabhat (INRIA, France, and IIT Delhi, India)

1 Introduction

A source level symbolic debugger is a tool that runs a program and presents a user with a virtual execution of the source code of the program. The user can stop the execution of the program, putting breakpoints or otherwise, and observe the value of variables at intermediate points. Also sometimes the execution may abort due to external interrupts at which point the user may like to see the status of various variables. Apart from observing, a conventional debugger also provides the facility to change the value of variables at runtime.

Optimization of the machine code is essentially done by all the compilers. The popular cycle of edit-compile-debug is normally followed with unoptimized code. However there are situations where this cycle needs to be done with optimizations on. Such situation may arise due to a bug which shows itself only on optimized code, or due to the resource limitation of the machine, or when rectifying a bug in an already crashed application whose core is available.

2 Problem

To develop a source level symbolic debugger for optimized code which has functionalities equivalent to a conventional debugger. The debugger should not in any case present the user with an apparently false value (inconsistent with the user's perception of source code) which forces her to deduce logical errors which are actually not present.

There are two inherent problems that optimization presents for debugging.

1. Code location problem
2. Data value problem

2.1 Code location problem

It concerns maintaining a mapping between source lines and machine instructions. Normally a statement in High Level Language corresponds to a series of instructions in the assembly code. In the unpotimized version the correspondence between these two is well known which is why the debugger works. However with optimizing transformations instructions are deleted, replicated, changed and rearranged (often across the statement boundaries). Thus with dissolving of statement boundaries the mapping between source line and instruction block becomes ill-defined. Now when the user sets a breakpoint it is not clear as to where to put the corresponding break in the assembly code.

Figure 11: Lost boundary in rescheduled code

Source Code	Unoptimized Object Code	Rescheduled Object Code
S1: d = f + g;	I1: load R1, f	I1: load R1, f
	I2: load R2, g	I2: load R2, g
	I3: fpadd R6, R1, R2	I5: store (R4), R3
S2: b = c * a	I4: fpmul R4, R5, R3	I4: fpmul R4, R5, R3
S3: *p = a	I5: store (R4), R3	I3: fpadd R6, R1, R2

2.2 Data-value problem

The value of source variables which the debugger retrieves and displays must be consistent with what the user expects with respect to the source statement where the execution has halted.

Due to optimizations, the state of the variable at a certain point must be one of the following.

Non resident: A variable V is said to non resident if the value in the register assigned to V may be the value of some variable other than V. This may happen due to global register allocation wherein the register assigned to V may be shared with other variables and temporaries.

Non-current: The set of variables whose values will be incorrect at a point are said to non current at that point in the computation. This may occur due to optimizations like code motion, dead code elimination etc. Non currency results due to tampering with the stores associated with that statement. The non current range of a variable is a section of straight line code such that the variable is non current at each statement in the range.

Initial Code	Optimized Code	Noncurrent ranges
1. A = B + C		
2. B = D + E	1'. G,B = D + E	| A
3. C = F + H	2'. C = F + H	| | G
4. G = D + E		|
5. A = 2 * B	3'. A = 2 * B	|

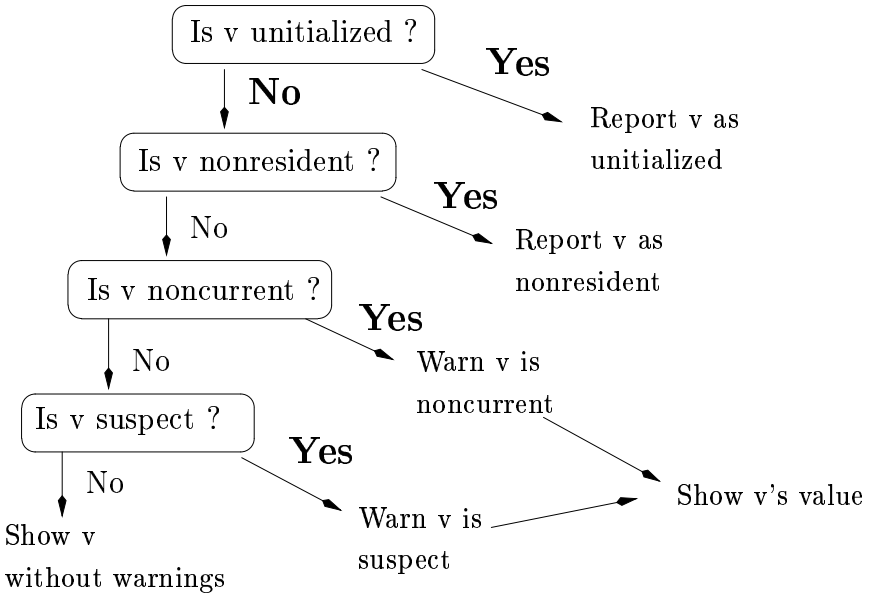
There are certain global optimizations which do not alter stores in to the variables either by eliminating or reordering them. Clearly application of such optimizations would not cause non currency and consequently the variable values in the new code will exactly match those in the original source program. These optimizations include constant folding, copy propagation, global common sub expression elimination, loop unrolling, strength reduction and code replication. Unfortunately they affect the ability of the user to insert

Figure 12: Noncurrent variables in locally optimized code.

Error occurs within	Is reported at	Noncurrent variables
1'	2	A
2'	3	A,G
3'	5	A

breakpoint. Though these optimizations themselves do not hamper debugging but they work as catalyst and promote other optimizations in successive iterations. For instance constant propagation or copy propagation may eliminate all uses of an assignment's left hand side, thus subjecting the assignment to elimination by dead code elimination.

Figure 13: Handling data value problems.



Actual value: If at a breakpoint a variable V is resident, then the value in the runtime location of V corresponding to some source level value of V is said to be the actual value of V.

Expected value: On the other hand, the value user expects V to have based on the context of the source breakpoint statement is the expected value of V.

Suspect: At times the debugger cannot determine whether the actual value of V conforms to expected value. In this case V can either be current or non current. Such situation arise due to multiple path reaching a breakpoint or an out of order execution of a pointer assignment. Pointers typically cause aliasing and at runtime a pointer variable may point to a host of variables. Determining precisely which variable the pointer is referring to is not always possible. Thus taking a conservative approach

anyone of the set of variables might be non current, and hence are suspect. The debugger ought to warn the user of the authenticity of all the suspect variable values.

Figure 14: Endangered variables at breakpoints in the code of previous figure

Breakpoint Object	Breakpoint Source	Source expression evaluated by Instruction	Nonresident Variables	Roll Forward	Roll backward	Suspect
I1	S1	f	b			
I2	S1	g	b			
I3	S3	*p = a	b	d		
I4	S2	b = c * a	b	d	*p	
I5	S1	d = f + g	p		b	f, g

Roll Forward Variable: At any point a variable V is said to be RFV if it should have been executed ideally but was not because the storage was delayed due to code sinking/dead code elimination. Consequently the variable at that point contains stale value.

Roll Backward Variable: These are variables which due to premature assignment exhibit a value they should attain at a later point in execution. This typically happens due to code hoisting.

The debugger should positively try to determine whether the actual value of V corresponds to the expected value, in which case V is current and the actual value is displayed without warnings. If a variable is found to be non current, suspect or current, the debugger has to convey to the user in source terms how optimizations have affected the value of a variable. In any case the debugger cannot allow display of a faulty value without appropriate warning.

3 Constraints

- The debugging should be non-invasive. Any change to the object code must be limited to the implementation of breakpoints. That is to say, the code generated by the compiler for debugging is the same as code generated otherwise.
- Dynamic data modification need not be supported by the debugger.
- The user should never be misled by receiving incorrect information from the debugger.
- The implementation should have minimal impact on performance and resource consumption

4 Approaches

4.1 Hennessy's (Ref. [1])

This is one of the first and foremost scheme used in this field. It is used for local optimization and uses augmented computation dag to represent the code dependencies in both locally optimized reordered code and the original unordered code. This however does not deal with arrays and pointers as dag based implementation of indirect variable references is not straightforward.

Salient features

- Each statement of the basic block is denoted as a dag.
- Constants and identifiers are leaves.
- Interior nodes denote operators.
- Each node has a code start label which specifies the start of generated target instruction sequence for the node.
- Each node has another label.

Figure 15: Structure of the label attached with DAG node

Variable name	Label type	Actual node pointer

Current: Variable is a current variable if the current value of the variable is the value of the node.

Old: Represents the old value of the variable. Represents the value of the labelling variable at an intermediate point of the block.

Actual node pointer: It is the dag node where the actual store into the labelling variable should have been performed in the unoptimized code.

- Each node denotes a storage in a variable.
- Each statement in the source program corresponds to a dag/subdag.
- Actual node pointer cannot point to a previous node
- Post order traversal of the dags in the increasing order of their code start label gives the affect of original code sequence at all statement boundaries.
- Breakpoints can be inserted only at
- **a)** Statement that has a starting node not shared by other statements.
  b) At a statement S whose shared starting node corresponds to a statement appearing only after S in the source code.

Figure 16: **Algorithm 1.** Detection of Non current Variables in Locally Optimized Code

Input: A dag D and a node d that is the error or breakpoint location.

Output: The sets RFV and RBV.

Method:

```
RFV := 0; RBV := 0;
{ First pass: find identifiers that should have been stored into }
for all nodes n < d do
  Fixed := { v | (v is a variable name in a current label on n)
                                  and  (node pointer of that label < d) }
for all nodes n element of D do begin
    if  (n < d) and (code start(n) > code start(d)) then
      { n should have been executed but was not }
      RFV := RFV ∪ { v | v is in a current label on n } ;
    { compute the RFV set that arises from eliminated stores }
    if n < d then
      RFV := RFV ∪ ( { v | v is in an old label on n } \ Fixed );
      { compute the RBV set }
        if  code start(n) < code start(d) then
          RBV := RBV ∪ { v | (v is in a current label on n)
                    and (actual node pointer of the label ¿= d) }
          else if  n > d then  RBV := RBV ∪ { v | v is in a current label on n }
          end
```

4.2 New Compiler Debugger Interface: CDI (Ref. [2])

This approach suggests to show actual behavior than expected behavior. instead of trying to imitate the unoptimized behavior it believes in presenting the true runtime behavior of the code. This is called visual debugging. The programmer is expected to have a thorough understanding of the optimizations performed so that he might figure out the semantics. The source granularity achieved in this approach is finer than statement level whereas in a usual debugger the user can figure out only up to a statement where the fault is. Here the syntactic structure, such as expression, variable etc. are highlighted in the process of execution.

On occurrence of a breakpoint, the text sections of the program corresponding to the currently executing code are highlighted.

The expression in the basic block which have been executed are highlighted in a different manner to distinguish them from the part yet to be executed.

The user can step through source and the respective expression(as in optimized code) would become prominent. Repetitive stepping will give rise to animation as successive section of the program are highlighted with the progress of execution. The programmer is supposed to work using this visual feedback.

The CDI requires:

- Extensive compiler support to generate the elaborate information required for debugging.
- Data files containing debugging information.

The usability of such a debugger would vary from user to user depending on the expertise he has in understanding the effects of each optimization. Possible outcome of the animation might be non sequential execution of code or unexpected change in the value of a variable.

Major advantage of CDI is that it provides finer control over the execution. For example, it allows step wise execution of 5 syntactical units than just a single statement.

4.3 Dynamic Deoptimization (Ref. [4])

Almost all the approaches mentioned up to now provides some limitation like it is not possible to recover value of all endangered variables, to single step during execution, or to change the value of a variable at runtime.

In the earlier system, optimization is given preference over debugging which leads to restricted debugging.

With dynamic deoptimization and interrupt points the kinds of optimization permissible becomes restrained. Interrupt points are the points in the code where the program can be interrupted. The state is guaranteed to be consistent here. The debugger can only be invoked at interrupt points. Consequently debug information need to be generated only for these points.

4.4 Machine Dependent Optimization(Instruction scheduling) (Ref. [9])

Normally, a high level statement generates a series of sequential instructions in machine code. However, to obtain gains from pipe lining the instruction scheduler rearranges them. Most of the optimizations are performed on medium/high level Intermediate Representation. But the scheduling can be performed only on low level representations as it is architecture specific. In this approach, annotations are added to the instructions to keep track of the changes performed and to enable recovery.

for each IR operation N, attach the following information
sched(N) : highest offset corresponding to N in object code.

for all expression A,
seq(A) : denotes the order in which A is executed in canonical order.

for all instruction I,
offset(I) : Position of I in the basic block

Attach the following information with each instruction

- Source_Reg(I),

- Dest_Reg(I)
- Address_Reg(I)

for all register R present in I attach
Last_DefI(R),
Next_DefI(R)

suspect variables are caused by
function calls.
indirect assignments (using ptr.) with non recoverable address expressions.

Note that variable that have been assigned register cannot be suspect variables as they remain unaffected by both indirect assignment and function calls.

Address Recovery :-
For all R element of Address_Reg(I):
(Last_DefI(R) < 0) $\vee$ (Next_defI(R) ≥ 0)

Assignment Recovery :-
(Offset(I) < 0) $\vee$ Next_DefI(I) ≥ 0)
for all R $\in$ Source_Reg(I):
(Last_RegI(R) < 0) $\vee$ (Next_DefI(R) ≥ 0)

4.5 Ali Reza Adl-Tabatabai and Thomas Gross's approach (Ref. [6])

This approach concentrates on the source level debugging of locally and globally optimized code and deals with endangered variables caused by these optimization.

It assumes that if the debugger can detect endangered variables caused by code hoisting and dead code elimination, then we have the foundation to debug optimized code, since these two transformations capture the effect of the elimination and movement transformations, which are root cause of endangered variables. Movement transformations can be either code hoisting or code sinking.

Code hoisting: This optimization copies an expression from a block B to one one more block that are post dominated by B.

Code sinking: This moves an expression E from a block B to one or more blocks that are dominated by B.

Detecting endangered variables caused by code hoisting:

Definition1: A redundant expression Er hoist reaches along a path P = < start,...., O), if there exists a hoisted expression Eh such that Er = RedCopy(Eh), and Eh reaches along P, and Er does not occur after the last occurrence of Eh along P.

Figure 17: **Algorithm 2.Finding the Subset of Noncurrent Variables that Are Recoverable**

Input: A dag D, a stopping point d, and the two sets of variables RFV and RBV.

Output: The set Recoverable is subset of (RFV ∪ RBV)
whose members are variables with recoverable values.

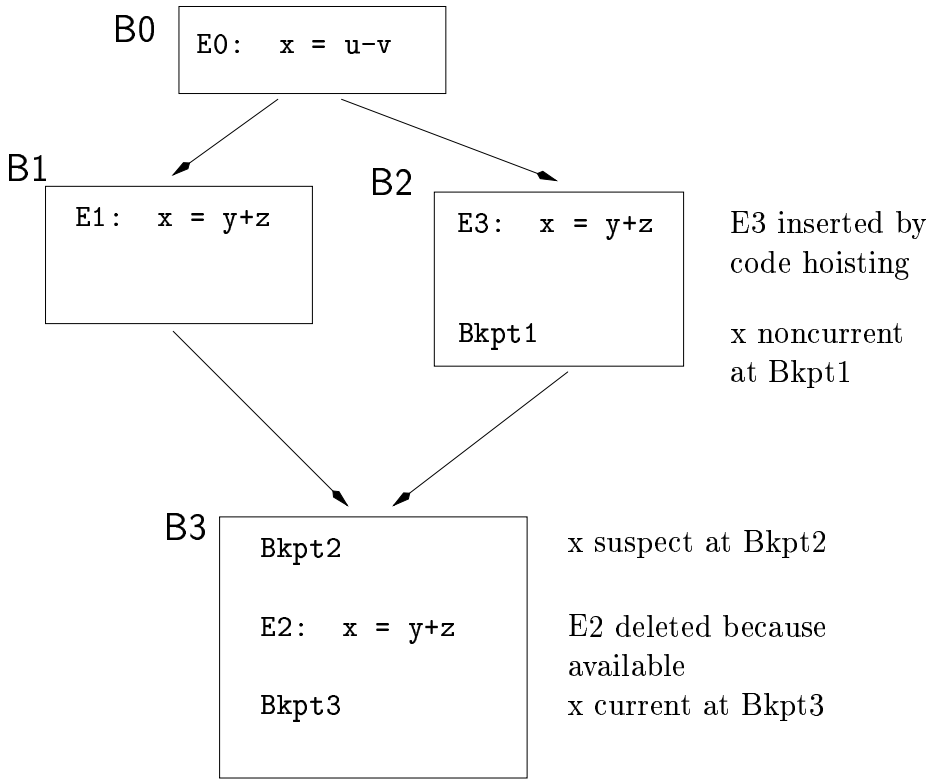
Function Employed: Available, defined as follows:

```
Available(n,d) = if there exists v (v is a variable in a current label on n)
                    and  (code start(n) < code start(d))
                 then  true
                   else if  n is a parent node
                   then  Available(right child, d)
                      and Available(left child,d)
                   else  { leaf node }
                     v := variable name in node n;
                     there exists p (p contains v in a current label
                         and  (code start(p) < code start(d)).
```

Method:

```
 Recoverable := 0;
 { First we attempt to recover all RBV variables }
 for  all v element of RBV do begin
    n := max { m E D | m has an old label for v } ;
    if  (n != null) and  Available(n,d) then  Recoverable := Recoverable ∪ { v }
    end
 { Recover RFV set }
 for  all v element of RFV do begin
    if  there exists n (v is in a current label on n and n < d)
      then  Recoverable := Recoverable ∪ { v }
      else begin
        n := max { m element of d | m contains an old label for v } ;
        if  n != null and  Available(n,d) then  Recoverable := Recoverable ∪ { v }
        end
```

Figure 18: Example of code hoisting.



Lemma1: Let Er be a redundant assignment expression that assigns to a variable V. If Er hoist reaches along a path P = < start,....,O) and P is the execution path traversed to a breakpoint B, then V is non current at B due to the premature execution of Er.

Lemma2: If Er hoist reaches along all paths leading to a point O, then at any breakpoint occurring at O, V is non current due to the premature execution of Er.

Lemma3: If Er hoist reaches long at least one but not all paths leading to a point O, then at any breakpoint occurring at O, V is suspect due to the possible premature execution of Er.

Detecting endangered variables caused by dead code elimination:

Definition2: A variable V is dead reaching along a path P = < start,...,O), if there exists a dead assignment Ed that assigns to V such that Ed occurs in P and no assignments to V occur along P after the last occurrence of Ed.

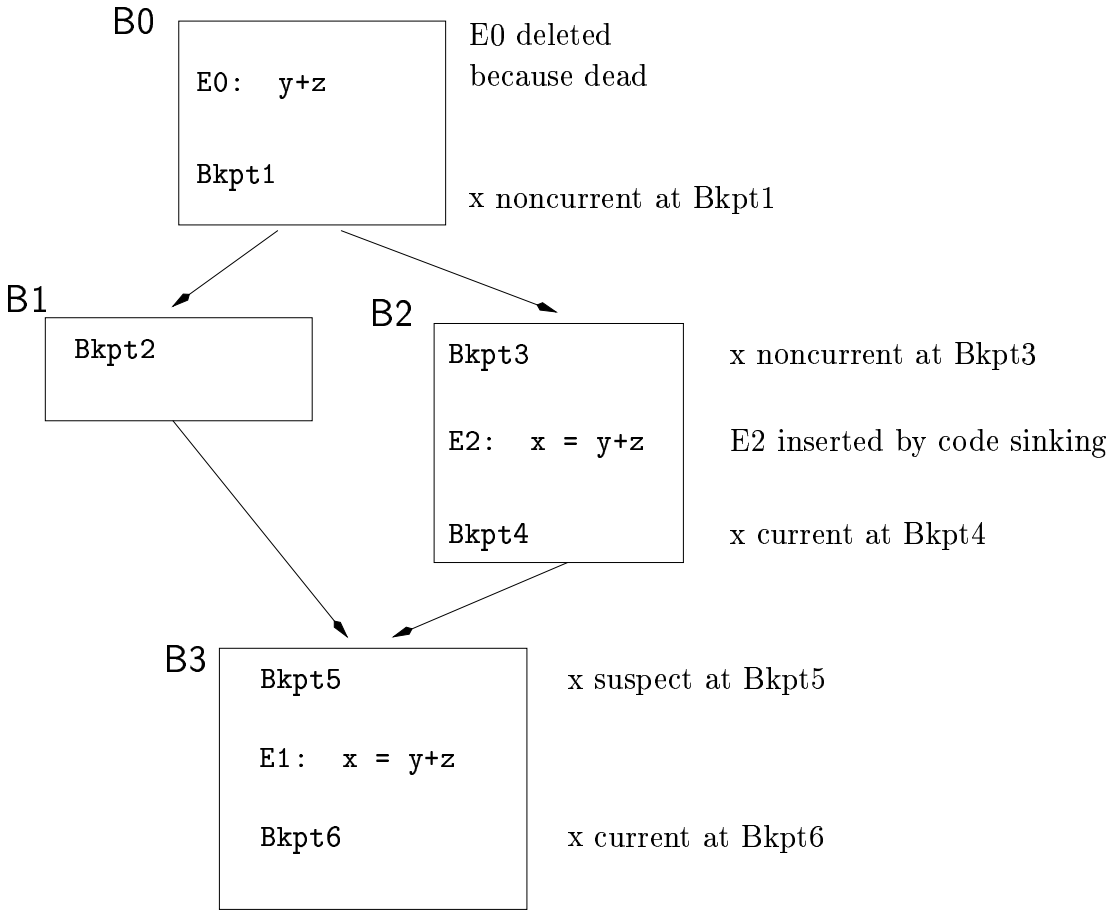
Lemma4: If a variable V is dead reaching along a path P = < start,...,O), and P is the execution path traversed to a breakpoint B, and V is not non current due to the premature execution of a redundant assignment, then V is non current at B because the actual value of V is stale.

Lemma5: If a variable V is dead reaching along all paths leading to a point O, then V is non current at any breakpoint occurring at O.

Lemma6: If a variable V is dead reaching along at least one but not all paths leading to a point O, then V is suspect at any breakpoint occurring at O.

For debugger analysis the compiler must perform extensive book-keeping to record the effects of optimization. The annotations record whether an operator was installed by optimization. Special IR marker nodes are also created for the use of debugger as required.

Figure 19: Example of dead code elimination.



5 Comparison between approaches

1. Hennessy's approach has been the first and foremost work in this field. Though it handles local optimizations adroitly, the graph model used for global optimization becomes too cumbersome to yield practical results.

2. The compiler debugger interface as presented by Brooks & Hansen is innovative. It presents the actual picture than presenting a facade to the unoptimized code. However the utility of the debugger hinges on the user's expertise in understanding the optimizing transformations. It represents an increased investment in compilation time and space requirements. The debugging scheme is not well defined. However, enhancements can be made by improving upon the GUI(visual effects).

3. Machine dependent optimization recovery schemes primarily deal with instruction scheduling. Maximum error recovery is done as long as only instruction scheduling comes into picture. How well the trick works, with effects of previous optimizations, is still doubtful. By and large it solves the problem, in its limited domain, effectively.

4. Dynamic deoptimization is yet another pragmatic solution proposed. Its' success rests on gap between interrupt points. This is invasive as contrary to the other approaches. The dynamic code generation has made it possible to get completely unoptimized behavior at invocation of appropriate procedures. The most popular practical debugger SELF has been made using the same technology. The space and time overheads are slightly higher as compared to undebugged code (about 122and 233

5. Tabatabai and Gross's approach asserts a rather strong statement claiming that all optimizations can be expressed as a combination of code hoisting and dead code elimination transformations or else they do not cause data value problems at all. Like almost every other approach, it also concentrates on data value problems only. Given the above assertion, their approach is simplicity itself.

6 State of the art

In spite of the fact that a lot of efforts have been put in this field for past two decades, the state of the art for support of debugging optimized code is generally quite poor. The few prominent debuggers available are CONVEX CXdb and HP9000 DOC systems. While CXdb detects non-resident variables (leaving the determination of variables' actual value to the user), DOC can detect variables that are endangered due to instruction scheduling. Unfortunately, it cannot deal with data value problems caused by global optimizations.

The SELF compiler is the first system to provide full source level debugging of globally optimized code. As mentioned earlier, it is based on the technique of dynamic deoptimization, with supports single stepping, running a method to completion, replacing an inlined method while only affecting the procedure activations that are actively being debugged.

7 Conclusion

Most of the debugging techniques developed deal only with data value problems. Code location problems are generally overlooked because they are less critical in misleading the user.

The inherent problem in debugging seems to stem from the cascading effect of different optimizations which are applied in various combinations a large number of times. To pass the accumulated effect of these optimizations to the debugger is extremely difficult for the compiler both in terms of computation and space overheads.

Of all the approaches seen so far, one can notice that they handle one or two optimizations at a time. A generic approach that can handle a host of optimizations is still lacking. Since developing a foolproof debugger seems quite difficult, the existing designs should aim to exploit the variance in the effects of various executions. Correspondingly, they should compromise the recovery of less frequent ones in face of the important ones. Thus, adequate performance levels can be achieved with acceptable complexity. For example, hoisting of assignment statements never occurs. Therefore, a debugger can take a conservative approach to detecting endangered variables caused by code hoisting.

References

[1] J. Hennessy. Symbolic Debugging of Optimized Code. *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3 (July 1982): 323-344.

[2] G. Brooks, G. Hansen, and S. Simmons. A New Approach to Debugging Optimized Code. *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, vol. 27, no. 7 (July 1992): 1-11.

[3] D. Coutant, S. Meloy, and M. Ruscetta. DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code. Proceedings of the *SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, Ga. (June 22-24, 1988): 125-134.

[4] U. Hülzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, Calif. (June 17-19, 1992) and SIGPLAN Notices, vol. 27, no. 7 (July 1992): 32-43.

[5] A. Adl-Tabatabai. *Source-Level Debugging of Globally Optimized Code.* Ph.D. Dissertation, Carnegie Mellon University, CMU-CS-96-133 (June 1996).

[6] T. Gross, A. Adl-Tabatabai. Source-Level Debugging of Scalar Optimized Code. 1996 ACM 0-98791-795-2/96/0005

[7] David Wall, Amitabh Srivastava and Fred Templin. A note on Hennessy's "Symbolic Debugging of Optimized Code". *ACM Transactions on Programming Language and Systems*, Vol. 7, No. 1, Jan. 1985, Pages 176-181.

[8] Ronald F. Brender, Jeffrey E. Nelson, Mark E. Arsenault. Debugging Optimized Code: Concepts and Implementation on DIGITAL Alpha Systems.

[9] A. Adl-Tabatabai. *Detection and Recovery of Endangered Variables Caused by Instruction Scheduling.* Carnegie Mellon University, 1993 ACM 0-89791-598-4/93/0006/0013.