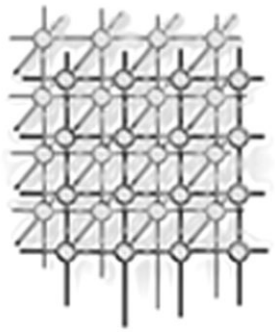
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2004; **16**:271–292 (DOI: 10.1002/cpe.774)



A fast and accurate method for determining a lower bound on execution time

G. Fursin[1,*,†], M. F. P. O'Boyle[1], O. Temam[2] and G. Watts[2]

[1]*ICSA, School of Informatics, University of Edinburgh, Edinburgh EH9 3JZ, U.K.*
[2]*Laboratoire de Recherche en Informatique, Bâtiment 490, Université Paris Sud, 91405 Orsay Cedex, France*

SUMMARY

In performance critical applications, memory latency is frequently the dominant overhead. In many cases, automatic compiler-based optimizations to improve memory performance are limited and programmers frequently resort to manual optimization techniques. However, this process is tedious and time-consuming. Furthermore, as the potential benefit from optimization is unknown there is no way to judge the amount of effort worth expending, nor when the process can stop, i.e. when optimal memory performance has been achieved or sufficiently approached. Architecture simulators can provide such information but designing an accurate model of an existing architecture is difficult and simulation times are excessively long. In this article, we propose and implement a technique that is both fast and reasonably accurate for estimating a lower bound on execution time for scientific applications. This technique has been tested on a wide range of programs from the SPEC benchmark suite and two commercial applications, where it has been used to guide a manual optimization process and iterative compilation. We compare our technique with that of a simulator with an ideal memory behaviour and demonstrate that our technique provides comparable information on memory performance and yet is over two orders of magnitude faster. We further show that our technique is considerably more accurate than hardware counters. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: memory performance; optimization tool; memory latency analysis

1. INTRODUCTION

Reducing the impact of memory latency by program restructuring can bring significant improvement to performance critical applications. Compiler memory optimizations, while attractive, are very limited because of the complexity of the memory and processor architecture. For reasons of tractable analysis,

*Correspondence to: G. Fursin, ICSA, Division of Informatics, University of Edinburgh, JCMB, King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, U.K.
†E-mail: g.fursin@ed.ac.uk

Received 20 September 2001
Revised 29 May 2002
Accepted 14 March 2003

compilers assume a rather simplified version of the memory hierarchy, viz registers and the first-level cache. Other components are considered in production compilers, but the model implicitly used is necessarily far less complex than the actual memory system which includes registers, different cache levels, TLB, write-buffers, buses and main memory. Furthermore, other features such as the interaction between the latest version of the operating system and the different architectural components are not addressed. As accurate, static, whole program interprocedural locality analysis is not currently available, aggressive optimization techniques are frequently restricted to a single procedure. Thus, overall, memory architecture conscious optimization of a program remains limited.

Therefore, whenever performance is at a premium, *manual program optimization* remains necessary. In practice, manual optimization is a trial and error process: a program transformation is applied, the program performance is evaluated, a new transformation is applied after analysis and so on. This process can be long and tedious, requiring possibly weeks of hand-tuning depending on the program and the performance goal. While this may be acceptable in certain academic grand-challenge projects, it is an expensive approach, particularly in industries where software development time dominates overall cost. At present it is not possible to determine the potential benefit to be gained from program optimization and whether or not the potential return is worth the investment.

This leads to the following technical question: can we find a way to estimate beforehand the potential benefit of memory program optimization? In other words, what will be the execution time improvement after the whole transformation process is performed? While it is difficult to provide an accurate value of the expected execution time beforehand, we can seek a lower bound on the execution time. Furthermore, if we can obtain this lower bound quickly, it may be computed at each step of the iterative manual optimization process and used to determine whether further optimization is worthwhile.

The lower bound on execution time of a program is defined here as a program with no cache misses. Memory parameters other than misses play an important role in program performance, but most optimizations target misses [1–3], so a lower bound which focuses on this criterion will be most useful in driving an iterative optimization process. Note that the minimum number of misses in a program is not zero (i.e. 'no miss') but the number of compulsory misses; however, the fraction of compulsory misses is almost always negligible compared with capacity and conflict misses [4,5].

Until recently, deducing the *no-miss* execution time from the normal execution time would have been relatively straightforward using hardware counters [6]: the execution time minus the number of misses (as recorded by hardware counters) times the latency would provide an accurate lower bound on execution time. However, superscalar processors now have non-blocking caches, out-of-order execution and complex memory hierarchies [7] which make it impossible to deduce the no-miss execution time based on the normal execution time and the number of misses. Memory access time may be completely overlapped with communication, hiding any memory latency. Conversely, memory stalls may sequentialize subsequent instructions, reduce ILP and have a much greater overall impact than the number of misses would indicate. This is empirically demonstrated in Section 5.6.

Processor simulators, like SimpleScalar [8], provide a simple means to compute this lower bound: it is trivial to modify a processor simulator so that it mimics a perfect cache behaviour. However, processor simulators have two drawbacks.

1. They only model the *processor* while the whole system can have a strong impact on memory performance: the way the TLB is reloaded, the bus arbitration mechanism, the physical to virtual mapping in lower cache levels, the type of memory (SDRAM, DDRAM), cache interferences

between several processes run concurrently and numerous other specific architecture-dependent issues. Consequently, we need a *system* simulator rather than a processor simulator. First, system simulators like SimOS [9] are, however, far less widespread and mature than processor simulators. Second, it is already very difficult to develop a processor simulator that accurately models an existing processor without privileged access to the processor internal workings [10], so that an accurate system simulator would require a huge effort to accurately model the chip set, the memory chips, the operating system and all other components.

2. A processor simulator is extremely slow: a simulated program on a current superscalar processor several hundreds times slower than the normal execution [8]. On a system simulator a 2000-fold slowdown or more is likely. Whether the simulator is used only once at the beginning of the optimization process or worse, at each step, such a slowdown is rarely acceptable for many programs and not tolerable for applications whose execution time exceeds a few minutes.

As we need to take into account the whole system architecture, and cannot afford excessive delays, simulators do not provide a satisfactory means for computing the execution time lower bound. In this article, we propose a technique that is both fast and reasonably accurate for estimating the execution time lower bound of a program, where we primarily focus on loop control structures and the array data structure. This technique has been implemented and tested on a range of programs and shown to be both fast and accurate.

This paper is structured as follows. Section 2 provides a motivating example and outlines the assembly modification technique used to determine the lower bound on memory access time. Section 3 provides a detailed description of our algorithm and is followed, in Section 4, by a brief description of the algorithm's implementation on two separate platforms. Section 5 provides as empirical evaluation where this technique is applied to the Spec FP benchmarks and two full applications. Section 5 also demonstrates how our lower bound estimation can be used to guide program optimization and compares its behaviour with respect to hardware counters and a simulator. Section 6 critically evaluates the technique and in Section 7 related work is briefly reviewed. Section 8 concludes the paper and outlines future work.

2. MOTIVATION AND EXAMPLE

This section provides a motivating example, illustrating the assembler modification technique to remove almost all cache misses without affecting the remainder of the program. The full algorithm to perform these actions is described in greater detail in Section 3.

The general approach is to modify the program so that it retains the characteristics of the original program but induces the minimal number of misses. Therefore, the execution time of the instrumented program will provide a lower bound on execution time of the original program once all cache misses have been eliminated.

In a program where loops and arrays dominate, almost all cache misses are due to array references within loops. The baseline of our technique is to transform each individual array reference into a scalar reference. The memory footprint of the resulting program would be negligible compared with the original footprint and the number of misses would be close to zero. The real challenge is to make sure that this transformation will not affect the rest of the program and its execution on a superscalar processor.

Let us consider the array reference `A[i]` in the following loop:

```
DO i = 1, N
  ... = A[i]
  ... = B[2*i+17]
  ...
ENDDO
```

After compiling on a Compaq Alpha EV6, this reference would be translated into the following assembly code:

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

where register \$19 contains the current target address of the load instruction, i.e. the base address of array `A` plus loop counter `i` times the size of one memory element (8 bytes in this example); `lda` is a misleading acronym, it is not a load instruction but an add instruction dedicated to address computations. So in this case, it increments register \$19 by 8 to fetch the next element of array `A`. The load instruction, `ldt`, fetches the data located at the address stored in register \$19 into register \$f13. These two instructions correspond to the array reference `A[i]`.

Assume now that we modify the `ldt` instruction as follows:

```
...
lda $19, 8($19)
ldt $f13, ($28)
...
```

where we substitute the register \$19 with register \$28 to access memory. Before executing the loop, register \$28 is set to a constant address which points to a memory address with preloaded data values that remain invariant throughout execution. Further `ldt` instructions within the loop still use register \$28 but change the offset to 8, 16 etc. Thus the reference to `B[2*i+17]` will be `ldt $f13, 8($28)`.

All the instructions are the same, the same number of computations is performed. But now the address referenced by each instruction `ldt` is constant over the whole loop execution. Consequently, the memory footprint of reference `A[i]` is reduced from $N \times 8$ bytes to just 8 bytes. Considering the minimum cache size is around 8 kbytes, and that the number of references is significantly less than a 1000 within do-loops, the memory footprint after transformation will almost always fit in cache and then only induce as many cold-start misses as the number of array references in a loop, which is negligible.

In the transformed code, all the instructions are the same as is the number of computations performed. Naturally, once the code has been transformed as above, it no longer executes correctly. Therefore, a copy is made of each program segment of interest at the assembler level and modified as described above. First, the instrumented segment is executed, then the original segment is executed to enable normal program execution. However, the instrumented segment can still modify variables so


```
...
br calc2_prep_          # Preparing data for transformed segment,
                        # and saving all registers

br calc2_tr_            # Executing transformed segment

br calc2_restore_       # Restoring registers

br calc2_               # Executing original segment
...
```

Figure 1. Backup, execution and restoration.

that the program may not run correctly afterwards. For this reason, *backup* and a *restore* procedures are added before and after the instrumented segment respectively.

For example, consider a subroutine `calc2` from the spec benchmark swim and the transformed assembler code shown in Figure 1, where `calc2_prep_` is the backup procedure, `calc2_tr_` is the instrumented segment, `calc2_restore_` is the restore procedure, `calc2_` is the original segment and `br` is the assembler instruction for branch and return. `calc2_prep` copies a minimal set of the data values accessed by the original segment into a new data area to be used by the modified program segment. In addition, all register values are saved and later restored. The transformed routine `calc2_tr_` is modified to refer to a greatly reduced number of data values residing in a special data area. Once this has executed the registers are restored to their earlier values `calc2_restore_`. Finally the original segment is executed `calc2_`. The state before `calc2_` is effectively that occurring in the original program. The only data changed is in an area not accessed by `calc2_` and the registers are restored to their correct values.

Ensuring that the lower bound of execution time is found without adversely affecting the control-flow and exception behaviour of the program requires careful consideration. The following section considers these issues and provides an overall transformation algorithm.

3. ALGORITHM

Figure 2 outlines the algorithm used to determine the lower bound of execution time. The first step simply profiles the entire program using a modified profiler collecting overall execution time and the execution time of subsections of the code—typically loop nests in our case. This allows later comparison and aids in evaluating the impact of memory latency. Step 2 is responsible for inserting calls after each memory reference to record the data values referred by the first execution of each load/store instruction. Step 3 executes this modified program collecting and storing the necessary data values. Step 4 is the main modification procedure. A duplicate copy of the appropriate routine is made. This copy is transformed so that the number of memory accesses is reduced to the smallest possible footprint while maintaining dependences and referring to valid data. Register save and restore routines are then inserted into the program. Once this has been achieved, the entire program is executed and the necessary profile data collected.

The following sections describe certain aspects of the algorithm in greater detail.

1. Profile original program on loop level
2. Instrument program segments to collect runtime data values and addresses
3. Run instrumented program
4. Transform program:
    - create copies of each segment
    - transform instructions with memory access inside loops so that they reference to preset values, analysing and keeping data dependences
    - allocate memory for preset values
5. Profile transformed program

Figure 2. Transformation algorithm.

3.1. Collecting data values

We wish to minimize references to memory in order to determine a lower bound on execution time. A naive approach would be to simply replace all load/store operations with NOOPs. However, this would alter the scheduling of the program and more importantly cause a large number of exceptions due to arithmetic on non-initialized register values. Alternatively, all load and store operations could refer to one initialized memory address that, after the first reference, would be permanently in L1 cache. Although reducing floating point exceptions, we have made every memory operation dependent on each other, completely changing the behaviour of the program. Our approach is to run the original program and gather the values of the data referred to by each memory operation on its first execution and then transform the program to always refer to these values. This dramatically reduces the footprint of the program; an array reference traversing N elements of an array will now just refer to the first element and reduces the likelihood of introduced exceptions as the references are to appropriately initialized values. When the program is later modified, the load and stores are now to these saved data values.

The gathering of data is achieved by inserting a jump to a data collection subroutine after each memory operation. Before jumping to the collection routine, the instruction number is pushed onto the stack, together with the memory address referred to:

```
instruction_no: load/store dest_reg, Mem[address]
                push instruction_no
                push address
                br collect
```

where `instruction_no` is simply the address in memory of the particular load/store instruction. Within the collection subroutine, the memory address and its value refereed to in the original memory operation are saved to two arrays; `addr` contains the `instruction_no` of the memory instruction plus the memory `address` referred to while `value` contains the actual value referred to,

i.e. `Mem[address]`. Only the first data value referred to by an instruction is stored and therefore a additional check array is used. The collection routine is of the following form:

```
if check[instruction_no] == 0
then
   check[instruction_no] = 1
   addr[next].ins_no = instruction_no
   addr[next].add = address
   value[next] = Mem[address]
   next = next + 1
```

This will collect all the necessary data but the additional overhead of jumping to a subroutine on every memory access is prohibitively expensive. It can increase the execution time by a factor of 15. Although still much faster than simulation, this is unacceptable for large applications. Instead we introduce self-modifying code, where we overwrite the original branch and push instructions with NOOPs once we have collected data for the first execution of any instruction. Thus, rather jumping to the collection routine each time a load/store is executed, it only takes place once.

The above code is therefore extended:

```
else
  Mem[instruction_no+word_size] = NOOP // overwrite 1st push
  Mem[instruction_no+2*word_size] = NOOP // overwrite 2nd push
  Mem[instruction_no+3*word_size] = NOOP // overwrite branch
endif
```

overwriting the two push and one branch instruction. This gathering of runtime data now only increases execution time by 15% on average. If two references are found to refer to the same memory address then this will be reflected in the transformed program so as to ensure that data dependences are preserved; this is considered in the next section.

3.2. Removing misses

Our technique maps all array reference into scalar references, reducing the memory footprint and the number of misses.

Once again, consider an array reference A[i] in the following loop:

```
DO i=1, N
   B[i] = ... A[i] ...
ENDDO
```

The assembler pseudo-code will be as follows:

```
Ri = 0
LABEL:
...
Addr_a = Addr_a0 + Ri * L_e ;
```

```
LOAD Rd, Mem[Addr_a]
Addr_b = Addr_b0 + Ri * L_e
STORE Rd, Mem[Addr_b]
Ri = Ri + 1
IF Ri < N go to LABEL
```

where Ri is an index register, Addr_a0, Addr_b0 are the addresses of the first elements of arrays A,B, L_e is the size of an array element and Rd is a data register.

We modify instructions

```
LOAD Rd, Mem[Addr_a]
...
STORE Rd, Mem[Addr_b]
```

as follows:

```
LOAD Rd, Mem[Addr_c1]
...
STORE Rd, Mem[Addr_c2]
```

where Addr_c1 and Addr_c2 are constant addresses which are set up before executing the loop and remain invariant throughout execution. They refer to the preloaded data in the array `value`. If the load instruction is stored at `load_ins_pos` then $Addr\_c1 = Addr\_value + x * L\_e$ where $addr[x].ins_{no} =$ `load_ins_pos` and *Addr_value* is the base address of the `value` array containing the saved data values. If the store instruction, `STORE Rd, Mem[Addr_c2]` is the next memory reference then $Addr\_c2 = Addr\_c1 + L\_e$ will refer to the next element of value. Thus the original reference to Addr_a and Addr_b have been replaced by access to the special data area values, `value[x]` and `value[x+1]`. The transformed code has the same instructions and the same number of calculations are performed, but now, however, Addr_c1 and Addr_c2 are constants over the whole loop execution and consequently the memory footprint of references A[i] and B[i] is greatly reduced. Normally, the assembler code for each procedure is traversed in order, with each assembler instruction containing a memory reference transformed to refer to the next array element in the saved data array `value`. However, it is important that the newly introduced addresses, i.e. Addr_c1 and Addr_c2, do not introduce cache conflicts. Due to the much smaller number of references in the transformed program, this is much easier to guarantee by padding the array where necessary.

3.2.1. Data dependences

In order to maintain the same data dependence structure of the original program, we attempt to ensure that if two memory accesses `Mem[Addr_a1]` and `Mem [Addr_a2]` are to the same memory address ($Addr\_a1 = Addr\_a2$) in the original code that this is preserved in the modified version. As it stands, each new memory instruction encountered will refer to the next element in the allocated data set. To overcome this we search through each procedure to see if one or more instructions refers to the same memory address. If so, we tag those instructions so that when the memory operands are modified they all refer to the same memory address. Thus, if there are two instructions residing in addresses *ins_no_i*, *ins_no_j*, *ins_no_i* ¡ *ins_no_j*, referring to the same memory address

addr[*ins_no_i*].*add* = *addr*[*ins_no_j*].*add* with their transformed memory address set to distinct values Addr_c1, Addr_c2, then Addr_c2 should be set to Addr_c1 to preserve data dependence. This is an $O(n^2)$ procedure where n is the number of instructions containing memory references in a procedure; it has a negligible impact on the execution time of the overall algorithm. This technique will only work if the dependences are still visible after removing misses. If there are runtime dependences or dependences within just a restricted part of the iteration space, we will not detect them. In practice, however, this has little impact on the accuracy of our scheme.

3.3. Correct code execution

The modified procedure refers to the data set gathered during step 2 and stored in the `value` array and will not affect the remaining procedures. The data area is preloaded with the correct values from the `value` array each time before executing the instrumented program segment. It may, however, affect register values so these are saved before execution and restored immediately afterward. Finally, the original unmodified segment is executed.

```
CALL preload_variables_and_save_registers
CALL instrumented_segment
CALL restore_registers
original_segment;
```

3.4. Array indirection and control flow

Array indirection frequently causes problems with static analysis due to compile-time undecidability. As the values for the indirection and main array are gathered during step 2, these values are saved and referred to later in the modified form of the program. Hence array indirection or other complex addressing such as tree structures etc., do not cause any difficulties.

Arbitrary control-flow, however, does cause problems. A conditional within a do loop whose value is dependent on an array element will be assigned to either true or false for the entire duration of the loop in our current approach. This is due to the first referenced value of the array being loaded each time for the entire loop. Alternatively, during the initial profiling in step 1, the number of times a particular branch is taken may be recorded and replicated to give a more accurate run. If there are distinct phases in which a particular control-path is taken, this may also be recorded. This is the subject of ongoing research.

4. IMPLEMENTATION

The assembler modification technique described above is ideally implemented in the code generation phase of a compiler. Due to the inevitable lack of access to the internals of the processor vendors' compilers we have implemented our algorithm as a post code generation, stand-alone assembler modification transformation. This algorithm works on assembler level and is independent of high level language.

To show the portability across platforms with different instruction sets we created a complete toolset for automatic analysis and instrumentation of codes for two platforms: the Compaq Alpha (21264, 500 MHz, RISC, Unix) and the Intel Pentium (Pentium II, 350 MHz, CISC, Windows 2000). These are both superscalar processors with out-of-order execution and have two levels of cache. The x86 and Alpha ISAs, however, are very different in structure and based on CISC and RISC design philosophies respectively.

4.1. Alpha

To build the instrumentation tool we needed to analyse the ISA and in particular, the LOAD and STORE instructions. Typical LOAD and STORE instructions in the Alpha assembler have the following format:

```
load_instruction  $f_data_register, offset($address_register)
store_instruction     $f_data_register, offset($address_register)
```

where load_instruction is 'lds' for loading long word, 1dt' for loading quad word, etc. and store_instruction is 'sts' for storing long word, 'stt' for storing quad word, etc.

In general, the saved data values should be referenced via a register dedicated to storing the global data pointer. With the Compaq compilers, however, register $28 is usually left free and therefore we use this to refer to our saved data values. Memory is allocated and assigned with specific preloaded data (i.e. the `value` array described above) and register $28 is assigned the base address of the `value` array.

4.2. Pentium

The task of transforming the assembler code is more difficult with the Pentium, as it uses a complex instruction set, where references to memory can be embedded within arithmetic instructions.

The typical instructions to be transformed have the format:

```
instruction ... type_of_word PTR
immediate_address+offset1[address_register_expression+offset2] ...
```

where 'PTR' indicates that this instruction has a memory access; type_of_word is 'DWORD' for loading or storing double word, 'QWORD' for loading or storing quad word, etc. The address part of the instruction may consist of an immediate address and its offset plus an address_register_expression and its offset, where the address_register_expression can be a complex linear expression such as register1+register2*const.

We have to transform a range of instructions containing memory references including 'mov' for moving data from/to memory, 'fld' for loading into floating point register and 'add' for addition.

Consider:

```
add edx, DWORD PTR __BLNK__[eax-56]
fld QWORD PTR __MEM__[eax+esi*4-60]
mov DWORD PTR __CALC__+1080, -1
```


To transform this piece of code we, once again, need to allocate memory, assign it with specific preloaded data and update all memory references. Due to the restrictions of the x86 CISC addressing mode this is best achieved by assigning all addresses with the immediate address of the allocated memory plus an offset:

```
add edx, DWORD PTR __SPECIAL_MEM__
fld QWORD PTR __SPECIAL_MEM__+8
mov DWORD PTR __SPECIAL_MEM__+16, -1
```

Thus our generic technique requires little platform specific modification even for radically distinct ISAs such as the Alpha and x86.

5. EXPERIMENTS

This section applies our technique to two platforms and a range of benchmarks determining the lower bound of execution time. The accuracy of our technique is then evaluated with respect to a processor simulator. This is followed by an evaluation of the technique in guiding manual and compiler based iterative program optimization. Our technique is, finally, compared with hardware counters and shown to be a more useful measurement of an execution time lower bound.

5.1. Experimental framework

The experiments were performed on a Compaq Alpha 21264, 500 MHz (four-way superscalar, out-of-order execution, 64 KB first-level cache, 4 MB second-level cache) and an Intel Pentium II, (350 MHz, superscalar, up to five instructions per cycle, out-of-order execution, 16 KB first-level cache, 512 KB second-level cache). We transformed the most time-consuming loops of the SPEC FP 95 programs with the reference dataset and measured their new execution time. Two further full applications were also considered: Gauge (solves equations of quantum chromo dynamics—provided by Edinburgh Parallel Computing Centre) and FLU3M (solves Euler and Navier–Stokes equations with real gas effects—provided by ONERA, The French National Aerospace Research Establishment).

5.2. Results

Tables I–IV present the original execution time and the average instructions per cycle (IPC) as well as the lower bound on execution time and maximum IPC found by our tool. Tables I and II show the results from the Spec benchmarks on the Alpha and Pentium respectively. Tables III and IV present the corresponding results from the two full applications. All timings are obtained by inserting directly in the assembler, low overhead/high resolution timers around the loops of interest. Data are shown for all of the instrumented loops and in the graphs shown in Figures 3 and 4, the overall potential speedup for each benchmark and application on both processors is presented.

We can see that the expected percentage of improvement varies considerably and that, in some cases, it is very high. For instance, the potential speedup for `wave5` on the Alpha is over four while for `mgrid` it is only 1.31. Obviously, the memory system has a significant impact on `swim` performance for both processors.

Table I. Upper bounds on Spec performance: Alpha.

Program	Procedure/loop #	Original time	Original IPC	Lower time	Upper IPC	Speedup
101.tomcatv	main_1	30.8	1.0	12.1	2.4	2.5
	main_3	24.9	0.3	5.2	1.6	4.8
	main_5	10.4	0.6	2.7	2.2	3.9
102.swim	calc1_1	20.5	1.0	9.3	2.3	2.2
	calc2_1	25.9	1.1	9.3	2.9	2.7
	calc3_1	24.1	0.9	6.4	3.2	3.8
103.su2cor	adjmat_1	3.9	1.4	1.6	3.4	2.4
	bespol_1	3.6	2.4	2.6	3.5	1.4
	matadj_1	4.0	1.4	1.7	3.4	2.3
	matmat_1	10.8	1.2	4.0	3.4	2.7
	sweep_2	3.5	0.6	0.7	3.1	4.9
107.mgrid	psinv_1	22.0	1.9	18.6	2.3	1.1
	resid_1	43.4	1.9	34.9	2.3	1.2
	rpj3_1	7.4	1.0	3.7	1.9	2.0
110.applu	buts_1	16.0	0.7	6.4	1.7	2.5
	jacu_1	12.9	0.9	5.2	2.3	2.4
	rhs_3	3.9	1.5	2.4	2.4	1.6
	rhs_4	4.1	1.5	2.6	2.3	1.5
125.turb3d	dfct_1	19.6	0.8	5.9	2.6	3.3
	dfct_2	11.0	2.0	6.6	3.4	1.6
	trans_1	8.1	2.6	7.8	2.7	1.0
141.apsi	hyd_1	4.2	0.5	1.3	1.5	3.2
	leapfr_2	3.2	0.5	0.7	2.5	5.6
	trid_1	4.0	0.6	2.8	0.9	1.4
	trid_2	3.8	0.5	2.3	0.9	1.7
146.wave5	parmvr_3	8.2	0.8	3.7	1.8	2.2
	parmvr_4	20.6	0.4	2.6	3.0	7.8
	parmvr_11	4.5	1.0	1.8	2.6	2.5

If we examine the results[‡] shown in Tables I–IV more closely we notice that there is variation among the procedures in terms of expected improvement. In most cases we see that the relative ordering of dominant execution time remains roughly the same after memory overhead is eliminated. This, however, is not always the case. In the case of `turb3d` on the Alpha, for instance, the relative ordering of routines reverses after the memory access overhead is eliminated. The possible improvement ranges from a factor of 7.8 in the case of the fourth loop of `parmvr` in `wave` to just over 1 in the case of the first loop of `trans` in `turb3d`.

[‡]All codes were compiled with -O5 optimization level except `flu3m_scalar` were -O4 optimization level was used due to some minor technical problems.

Table II. Upper bounds on Spec performance: Pentium.

Program	Procedure/loop #	Original time	Original IPC	Lower time	Upper IPC	Speedup
101.tomcatv	main_1	59.8	0.8	41.4	1.1	1.5
	main_3	43.1	0.3	21.3	0.6	2.0
	main_5	31.1	0.2	7.5	0.7	4.1
102.swim	calc1_1	62.4	0.6	47.1	0.8	1.3
	calc2_1	62.3	0.5	25.0	1.1	2.5
	calc3_1	94.0	0.3	18.6	1.4	5.1
103.su2cor	adjmat_1	15.7	0.6	6.3	1.3	2.5
	bespol_1	11.9	0.4	5.7	1.1	2.1
	matadj_1	17.4	0.5	6.4	1.3	2.7
	matmat_1	42.4	0.5	15.3	1.3	2.8
	sweep_2	10.4	0.2	1.2	1.4	8.7
107.mgrid	psinv_1	76.6	0.6	49.4	0.4	1.6
	resid_1	159.7	0.6	90.3	0.4	1.8
	rpj3_1	15.3	0.4	6.1	0.5	2.5
110.applu	buts_1	54.8	0.7	30.8	1.3	1.8
	jacu_1	38.5	0.5	18.7	1.0	2.1
	rhs_3	10.7	0.7	8.6	0.9	1.2
	rhs_4	11.4	0.7	8.4	0.9	1.4
125.turb3d	dfct_1	57.6	0.3	12.4	1.7	4.7
	dfct_2	26.5	0.8	13.1	2.0	2.0
	trans_1	14.1	1.3	13.6	1.3	1.0
141.apsi	hyd_1	6.0	0.7	5.9	0.7	1.0
	leapfr_2	5.3	0.2	1.0	1.7	5.3
	trid_1	10.0	0.3	9.1	0.4	1.1
	trid_2	9.1	0.2	8.6	0.2	1.1
146.wave5	parmvr_3	30.9	0.2	5.9	1.3	5.2
	parmvr_4	52.3	0.2	9.8	0.9	5.3
	parmvr_11	14.5	0.4	4.0	1.4	3.6

Table III. Upper bounds on two applications' performance: Alpha.

Program	Procedure/loop #	Original time	Original IPC	Lower time	Upper IPC	Speedup
gauge	hm_3by3 1	6.0	1.0	2.9	2.0	1.6
	hh_lg3by3	5.2	1.3	3.1	2.0	2.0
	mm_3by3 1	4.7	1.3	2.8	2.0	1.7
flu3m	invtri3_4	3.0	1.4	1.9	2.5	1.9
	invtri3_5	1.0	0.9	0.5	2.0	3.3
	invtri3_6	3.6	1.4	2.1	2.0	4.8
	invtri3_7	1.5	0.8	0.8	1.7	1.2
	invtri3_10	2.3	0.9	0.7	3.4	3.2

Table IV. Upper bounds on two applications' performance: Pentium.

Program	Procedure/loop #	Original time	Original IPC	Lower time	Upper IPC	Speedup
gauge	hm_3by3 1	20.1	0.7	13.0	1.0	1.5
	hh_lg3by3	20.6	0.7	13.4	1.0	1.5
	mm_3by3 1	16.8	0.7	10.0	1.3	1.7
flu3m	invtri3_4	9.3	0.8	6.7	1.1	1.4
	invtri3_5	2.6	0.7	1.1	1.7	2.4
	invtri3_6	11.9	0.7	7.5	1.1	1.6
	invtri3_7	2.7	0.5	1.1	1.1	2.5
	invtri3_10	6.3	0.4	2.1	1.1	3.0

This information is of particular use in determining where to expend effort in optimizing a program. If we consider `tomcatv` on the Alpha, the first loop dominates time, but the amount of performance improvement available is greater for loop 2 than loop 1: 19.7 versus 18.7.

If we now consider the two full applications, we can see that the opportunity for improvement ranges from 1.2 to 4.8 across the dominant routines. `Gauge` is a highly tuned application, yet in most cases a 50% improvement is available if the cost of memory latency could be amortized on the Alpha. If we compare the results of the Pentium with those of the Alpha the overall original IPC is less on the Pentium and the potentially available speedup is also less. This could be due to a more efficient compiler on the Pentium able to amortize the cost of memory or a facet of its CISC architecture.

5.3. Performance validation

To fully validate the fact that the instrumentation only affects memory behaviour and that the lower bound can effectively be interpreted, we have performed an additional experiment using a full processor simulator. Using *SimpleScalar* [8] we modelled a superscalar processor with similar characteristics as the Alpha EV6 (however, this is not an accurate model of the EV6, we can only state that it has roughly the same characteristics). We modified the simulator so that the cache and the TLB are perfect, i.e. all memory requests hit in the first-level cache and the TLB. Then, we have run both the original and the instrumented SWIM code on this simulator. Since the memory system is perfect, if both programs differ only in terms of memory behaviour, their performance on the processor simulator with a perfect cache should be nearly identical. More exactly, we compared the performance of the instrumented routines only in the instrumented version (ignoring the backup, restore and original routines) with all the routines of the original version. The results in Tables V and VI, confirm that instrumentation barely affects the overall program behaviour. The IPC of the transformed program, 3.02, is very near that of the original program when simulated on a machine with perfect cache 2.98. Table VI shows the memory performance when the two programs are run on the same simulator but with a normal cache. Here we can see that the difference between both programs is high both in terms of IPC and cache/TLB miss ratio. Furthermore, the transformed program is shown to have ideal memory behaviour having a zero miss ratio.

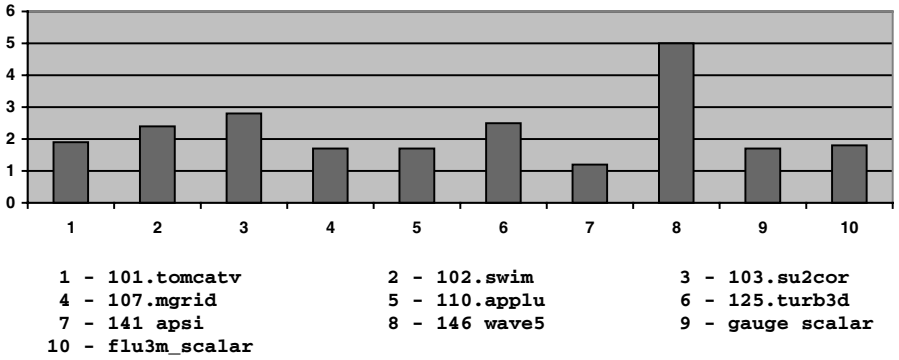


Figure 3. Maximum potential speedup Pentium.

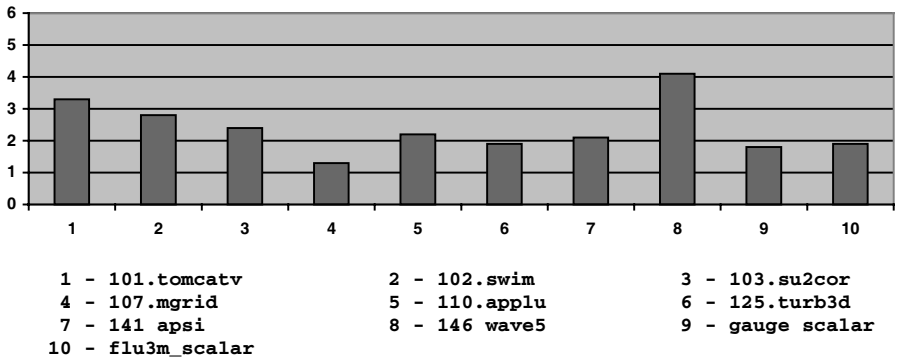


Figure 4. Maximum potential speedup Alpha.

Table V. Instructions per cycle (IPC).

	Original program	Transformed program
Normal cache	2.42	3.02
Perfect cache and tlb	2.98	3.02

Table VI. Cache behaviour (normal cache).

	Original program	Transformed program
Number of L1 accesses	295 705 805	288 213 871
L1 miss ratio	7.22%	0.0%
Number of TLB accesses	295 705 805	295 738 993
TLB miss ratio	0.2%	0.0%

Table VII. Execution time in seconds.

Original	Manually optimized	Lower bound
106.31	69.30	29.61

Table VIII. Execution time and hardware counter miss rate for the matrix multiplication kernel: Pentium.

	Time (s)	L1 cache misses	L2 cache misses
Original	96.5	0.448	0.415
Lower	11.5	0	0
Library	11.9	0.033	0.002

5.4. Guiding optimizations

We examined one program, SWIM, in more detail and attempted to optimize it by hand in order to verify that there was further program performance available as indicated by our analysis. We applied a number of memory conscious transformations including: blocking, padding, loop merging and forward substitution. The results in Table VII shows the execution time of the original program, the manually optimized program, and the lower bound.

These results show that additional improvements are available and may be partly achievable by manual optimization. Whether the lower bound on execution time is always achievable is not clear; the technique provides an estimate of the potential gain, allowing prioritization of effort. Thus, while the lower bound may not always be an achievable minimum, the ratio between the lower bound and the original can be used as a measure of optimization potential.

5.5. Iterative compilation

The current technique can also improve the efficacy of iterative compilation [11]. The goal of iterative compilation is to explore a large optimization space of the application without knowledge of the target machine, and to find out the optimal optimization sequence within that space. In practice, however, since the number of transformations is potentially infinite, the transformation space is necessarily restricted [11]. Using the method described in this article, we may target those sections of the application program which have the potential to be improved so as to reduce the transformation space to be searched.

Consider the two Spec benchmarks SWIM and MGRID and the potential speedups available on the Alpha as shown in Figure 4. SWIM has a potential 2.8 times speedup while in the case of MGRID, the best available speedup is a 20% improvement. Using an iterative compilation techniques as reported in [11], it is possible to achieve a 40% improvement on SWIM but only 10% on MGRID, reflecting the respective potential improvements shown in Table VII.

For further evaluation of the lower bound as a guide to in iterative compiler optimization we executed a matrix multiplication kernel on the Pentium whose execution time, 96.5 s, is shown in Table VIII in the row labelled original. The lower bound of execution time provided by our tool is 11.5 s—almost nine times faster than the original. When we applied iterative compilation to this routine we were able to reduce the execution time from 96.5 s to 19.8 s—almost five times faster. Now the question is, could we do better or is the lower bound unachievable and the iterative solution the best possible? As this is a well known kernel, the manufacturer provides a tuned library whose execution time is also shown in Table VIII. This value, 11.9 s, is almost equal to the lower bound. Thus, at least in this example, the lower bound provides a realistic lower bound target for optimizing compilers.

5.6. Hardware counters

It may be argued that hardware counters [6,12] can give the same information provided by our technique with less effort. Simply determine the overhead due to memory access time, subtract this from the original time and this will give the lower bound on execution time. For in-order processors the formula for CPU execution time would be CPU execution time = (CPU clock cycles + memory stall cycles) $*$ Clock cycle [4]. Current out-of-order execution superscalar processor can considerably overlap CPU time and memory stall time, invalidating this formula. Furthermore, the impact of memory accesses can be severely underestimated by hardware counters. To show this, we obtained the totals of all data references through hardware counters available on the Pentium for the matrix multiplication kernel and determined the memory access overhead using the following equation [4].

$$\begin{aligned}\text{Memory access overhead} = \text{Data references} * (&\text{HitRate L1} * \text{HitTime L1} + \text{MissRate L1} \\ &* (\text{HitRate L2} * \text{HitTime L2} + \text{MissRate L2} * \text{HitTime MainMemory}))\end{aligned}$$

There were 0.9×10^9 data references and the average hit times were measured as follows: HitTime L1 = 4 ns; HitTime L2 = 34 ns; HitTime MainMemory = 228 ns. Note that these figures depend on the processor and system configuration. Given the miss rate in Table VII, we can substitute these

values to give

$$\begin{aligned}&\text{Memory access overhead}\\&\quad = 0.9 \times 10^9 * ((1 - 0.448) * 4 + 0.448 * ((1 - 0.415) * 34 + 0.415 * 228)) * 10^{-9}\\&\quad = 48.2 \text{ s}\end{aligned}$$

Using the simplified equation above leads to CPU computation time of $96.5 - 48.2 = 48.3$ s. Note that this assumes that there is no overlapping of memory access and computation time and thus the CPU computation time could be greater.

If we remove all the cache misses so that all accesses are to L1 cache, the memory stall time will be 3.6 s.

Thus our lower-bound calculated from hardware counters is $48.3 \text{ s} + 3.6 \text{ s} = 51.9.5$ s. This, however, is 4.5 times higher than the time of the highly tuned code and our lower-bound (51.9 s versus 11.5 s). This factor of 4.5 difference is due to sequentialization of instructions in the presence of memory stalls reducing ILP. Thus, while hardware counters are useful in providing qualitative information about memory usage in different parts of a program, they do not provide accurate quantitative information and cannot be relied upon to give an accurate lower bound on execution time.

6. CAVEATS

The techniques developed in this paper raise several issues that may impact their successful implementation.

First, the current technique determines the lower bound on execution time assuming that a particular branch is always take, which is not the case. 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 only partly satisfactory, since the instrumented code instructions are not identical to those of the original code. Furthermore, if the control behaviour of the program is highly context sensitive, our analysis will be more limited in its use. Fortunately, a significant body of program loops in Fortran programs do not contain conditional branch instructions that depend on a value computed within the loop body.

Second, superscalar processor architectures can include load/store queues that are designed to avoid consecutive accesses to identical memory addresses. For instance, if a load to address `A` is issued after a store to `A` was performed and the store instruction is still in the queue, the load can directly access the data and avoid a memory reference. Our technique would potentially increase the number of such bypasses since, in a loop, an array reference is replaced by a reference to a constant so that the corresponding address is referenced many times. As a consequence, with such queues, our technique would provide an optimistic lower bound. However, another related property may partially compensate this bias: in the instrumented programs there may be more load/store dependences than in the original program, since the overall number of addresses used is much smaller but the number of load/store instructions is the same. Such dependences can degrade the exploitation of ILP, in which case the lower bound would be less optimistic than initially thought. Both effects should be investigated and evaluated further.

7. RELATED WORK

Optimizing compilers largely focus on exploiting instruction level parallelism and minimizing memory latency when analysing and optimizing uni-processor applications. Here we focus on issues related to memory latency. Typically, compilers employ a static approach whereby the program is analysed and a series of transformations applied which hopefully will reduce the memory access time overhead. Here there is an 'ideal form' of the program which the compiler attempts to obtain. The benefit of this ideal form is given the original source program, one can directly construct the transform that maps source to target [13]. In order to achieve this directness, an extremely simple model of what constitutes an ideal target is assumed, e.g. stride-1 access in a loop nest [13]. Furthermore, the effect of other transformations is not considered. Thus, rather than having an objective minimum to aim for, static techniques target a derived metric, for which there is no guarantee that this will actually benefit program performance.

Wolf *et al*. [14] have described an alternative approach that searches for an optimization by considering a restricted optimization space. Han *et al*. [15] also describe a compiler that searches for tile and pad sizes. Both of these compilers use static cost models to evaluate the different optimizations. Rather than using simply static models, several approaches to feedback-directed optimization have been put forward [16] based on runtime information. Dynamic information gathered off-line has been used in a limited sense for low-level compilers in, for example, the creation of superblocks [17] or hyperblocks [18] to enable efficient scheduling for ILP processors. These techniques are currently being employed in commercial compilers [19]. Profiles are also used to identify runtime constants that can be exploited at runtime [20]. For an excellent review of this work in this area see [16].

Regardless of whether dynamic or static analysis is used, none of the above approaches has explicit knowledge of the actual overhead due to memory costs nor to what extent it has been successful in reducing memory latency overhead. There are several possible approaches to the problem of defining a program's execution time lower bound which can be incorporated into a compiler or used by an expert programmer.

Several previous studies on optimal memory management [21–24] rely on Belady's optimal replacement algorithm [25], 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. Abraham and Sugumar [21] use Belady's algorithm to accurately characterize the notion of capacity and conflict misses. Burger *et al*. [22] use Belady's algorithm to define traffic inefficiency as the ratio between cache traffic and a perfectly managed cache using Belady's MIN algorithm.

Other approaches attempt to compute the number of cache misses due to a program in order to feed this information to a compiler or a programmer. The goal is not to compute an execution time lower bound but rather to provide a symbolic expression of the number of misses or a fast algorithm for computing the number of misses. Coleman and McKinley [2] propose a method for computing the optimal block size of tiling algorithms which is based on an estimate of cache misses in a tiled loop. Sanchez *et al*. [26] propose a combined static and dynamic method for estimating the number of misses in numerical do-loops. Ghosh *et al*. [27] propose to compute exactly the number of misses using *cache miss equations* and to use these equations to drive optimization techniques like padding or blocking. While these techniques provide useful information to the compiler, the implicit architecture is extremely simple: they assume a single-level cache hierarchy and ignore the rest of the

memory hierarchy and processor architecture. Consequently, they can considerably mislead a compiler or a programmer in the task of reducing the *execution time* of a program. This theme of statically determining the number of cache misses is further developed in [28]. Here static analysis based on an approximate cache is used to determine the number of cache misses. At best, this approximates the number of cache misses detected by the hardware counters and yet, as shown in Section 5.6, knowing the exact number of cache misses does not provide any useful information as to the lower bound on execution time for a superscalar machine.

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 like the *SimpleScalar* toolset [8], which provides a detailed cycle-accurate model of a superscalar processor. Such simulators can be easily used to determine the optimal memory performance by modifying the cache hierarchy so it appears to behave perfectly (no miss). However, these simulators model abstract machines (a generic superscalar processor) so that it is difficult to deduce from these experiments the true impact of memory performance on program execution time on real machines. Developing an accurate processor simulator without privileged access to a precise architecture description is a difficult and error-prone task. Current simulators have been shown to exhibit average errors of 40% [10], which makes them hardly suitable for driving an optimization process. Moreover, even very accurate processor simulators ignore or simplify the behaviour of many hardware system components so that the results provide a very rough approximation of the real execution time, particularly with respect to memory behaviour. Only system-level simulators that take into account the processor, the full hardware system and the operating system can be relied upon, like *SimOS* [9]. But such simulators induce program slowdowns that can easily exceed a factor of 2000, so they are not suitable for the iterative trial-and-error process of program optimization.

8. 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 lower bound is 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 significantly faster than simulation since the instrumented program execution time is at most double the execution time of the original program compared with a 500–2000 times slowdown for simulation-based techniques.

We have further demonstrated that our technique is accurate in that it is comparable to a cycle accurate simulator and that it provides a more accurate lower bound on execution time than using hardware counters.

While this technique provides a program execution time lower bound (a cache/TLB miss lower bound) it does not guarantee whether or not this lower bound can be achieved, although it allows us to determine those application which have memory problems and which are candidates for memory optimizations. In addition, we have shown, in one example, that the lower bound of execution time can be achieved where there is sufficient effort expended, as is the case for vendor supplied libraries.

Future research will focus on trying to define tighter lower bounds that are close to what can be effectively achieved through classic program transformation techniques. We will also investigate the

use of this lower bound calculation in reducing the transformation space for iterative compilation approaches to program optimization.

ACKNOWLEDGEMENT

This research was part of the European Esprit project MHAOTEU (Memory Hierarchy Analysis and Optimization Tools for the End-User).

REFERENCES

1. Wolf ME, Lam MS. A data locality optimizing algorithm. *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, June 1991; 30–44.
2. Coleman S, McKinley KS. The tile size selection using cache organization and data layout. *ACM SIGPLAN'95 Conference on Programming Languages Design and Implementation*, 1995.
3. Rivera G, Tseng C-W. A comparison of compiler tiling algorithms. *Computational Complexity*, 1999; 168–182.
4. Hennessy JL, Patterson DA. *Computer Architecture: A Quantitative Approach* (2nd edn). Morgan Kaufmann: San Francisco, 1996.
5. McKinley KS, Temam O. A quantitative analysis of loop nest locality. *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, October 1996.
6. Anderson JM, Berc LM, Dean J, Ghemawat S, Henzinger MR, Leung S-TA, Sites RL, Vandevoorde MT, Waldspurger CA, Weihl WE. Continuous profiling: Where have all the cycles gone?. *IEEE Transactions on Computer Systems* 1997; **15**(4):357–390.
7. Kessler RE, McLellan EJ, Webb DA. The alpha 21264 microprocessor architecture. *Proceedings of International Conference on Computer Design*, Austin, TX, December 1998.
8. Burger DC, Austin TM. The simplescalar tool set, version 2.0. *Computer Architecture News* 1997; **25**(3):13–25.
9. Rosenblum M, Bugnion E, Devine S, Herrod SA. Using the simos machine simulator to study complex computer systems. *Modeling and Computer Simulation* 1997; **7**(1):78–103.
10. Desikan R, Burger D, Keckler SW. Measuring experimental error in microprocessor simulation. *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Göteborg, Sweden, June 2001.
11. Kisuki T, Knijnenburg PMW, O'Boyle MFP. Combined selection of tile sizes and unroll factors using iterative compilation. *Proceedings of PACT'2000, Parallel Architectures and Compiler Technology*. IEEE Press: Philadelphia, PA, 2000; 237–246.
12. Dean J, Hicks JE, Waldspurger CA, Weihl WE, Chrysos GZ. Profileme: Hardware support for inst.-level profiling on out-of-order. *Proceedings 30th Annual International Symposium on Microarchitecture (TC-MICRO and ACM SIGMICRO)*. IEEE Computer Society: Research Triangle Park, NC, 1997.
13. Li W, Pingali K. Access normalization: Loop restructuring for NUMA compilers. *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992; 285–295.
14. Wolf ME, Maydan DE, Chen D-K. Combining loop transformations considering caches and scheduling. *International Journal of Parallel Programming* 1998; **26**(4):479–503.
15. Han H, Rivera G, Tseng C-W. Software support for improving locality in scientific codes. *Proceedings of CPC2000*, 2000; 213–228.
16. Smith MD. Overcoming the challenges to feedback-directed optimization. *Proceedings of Dynamo'00*, 2000.
17. Hwu W *et al*. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing* 1993; **7**(1/2):229–248.
18. Mahlke SA, Lin DC, Chen WY, Hank RE, Bringmann RA. Effective compiler support for predicated execution using the hyperblock. *Proceedings of MICRO 25*, 1992.
19. Cohn R, Lowney PG. Feedback directed optimization in Compaq's compilation tools for Alpha. *Proceedings of the 2nd Workshop on Feedback Directed Optimization*, 1999.
20. Mock M, Berryman M, Chambers C, Eggers SJ. Calpa: A tool for automating dynamic compilation. *Proceedings of the 2nd Workshop on Feedback Directed Optimization*, 1999.
21. Abraham SG, Sugumar RA. Efficient simulation of caches under optimal replacement with applications to miss characterization. *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, 1993.

22. Burger D, Kägi A, Goodman JR. Memory bandwidth limitations of future microprocessors. *Proceedings of the 23rd ACM International Symposium on Computer Architecture*, Philadelphia, PA, May 1996.
23. Burger DC, Kagi A, Goodman JR. The declining effectiveness of dynamic caching for general-purpose microprocessors. *Technical Report 1261*, University of Wisconsin, Madison, April 1996.
24. Temam O. Investigating optimal local memory performance. *Technical Report 97/25*, PRiSM, Versailles University, December 1997.
25. Belady LA. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* 1966; **5**(2):78–101.
26. Sánchez FJ, González A, Valero M. Static locality analysis for cache management. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT-97)*, San Francisco, CA, November 1997.
27. Ghosh S, Martonosi M, Malik S. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems* 1999; **21**(4):703–746.
28. Syzmanski B, Kaplow W. Program optimization based on compile-time cache performance prediction. *Parallel Processing Letters* 1996; **6**(1):173–184.

