On Increasing Architecture Awareness in Program Optimizations to Bridge the Gap between Peak and Sustained Processor Performance – Matrix-Multiply Revisited

David Parello
HP, France
& LRI, Paris South University, France

Olivier Temam
LRI, Paris South University, France

Jean-Marie Verdun
HP, France

Abstract

As the complexity of processor architectures increases, there is a widening gap between peak processor performance and sustained processor performance so that programs now tend to exploit only a fraction of available performance. While there is a tremendous amount of literature on program optimizations, compiler optimizations lack efficiency because they are plagued by three flaws: (1) they often implicitly use simplified, if not simplistic, models of processor architecture, (2) they usually focus on a single processor component (e.g., cache) and ignore the interactions among multiple components, (3) the most heavily investigated components (e.g., caches) sometimes have only a small impact on overall performance. Through the in-depth analysis of a simple program kernel, we want to show that understanding the complex interactions between programs and the numerous processor architecture components is both feasible and critical to design efficient program optimizations.

I. INTRODUCTION

To accommodate a constantly increasing clock frequency, computer architecture needs to be increasingly complex (long pipelines, cache hierarchy, branch prediction, trace cache, hardware prefetching,…) and it is increasingly difficult for compilers to generate programs that take full advantage of these architectures. As a result, the gap between peak performance and sustained performance rapidly increases. Bridging this gap is bound to become a critical technical and economic issue, highlighting the need for more efficient compiler program optimizations, or techniques and tools to assist end-users in optimizing their programs.

There is already a tremendous amount of literature on program optimizations and especially loop nest optimizations [1],[2],[3],[4],[9],[10],[11],[15]. Though a large fraction of this research work aims at improving program behavior on processor architecture, most of this research work actually target one specific architecture component like the cache [2],[3],[10],[18], the registers [23] or less frequently the TLB [24], but few attempt to consider *several* architecture components *together* and their possible interactions. And optimizations focused on one processor component that ignore most or all other processor components are likely to be much less effective than expected, if not totally useless.

Besides, most research work on compiler optimizations for processor architectures use *simplified models* (which are more tractable) of the different architecture components, *ignoring many architecture phenomena* and resulting in poor performance improvements. For instance, studies on cache conflicts started [2] way after caches were introduced [6], and only recently, Malik et al. [7] proposed a reliable method for taking into account the cache mapping function and addressing cache conflicts at compile-time.

More perversely, certain research topics on program optimizations for processor architecture are considered almost closed, e.g., memory optimizations, simply because we have been working on them for a long time. However, because of the rapidly increasing processor complexity, program behavior on processor architecture changes as well, and program optimizations need to be revisited accordingly or the gap between peak and sustained performance will keep increasing. One of the goals of this article is to highlight this issue and to show the task is tractable, provided program optimizations become more "architecture-aware", i.e., rely on much more detailed architecture models. More precisely, we want to show that:

- it is possible to achieve large performance improvements, and sometimes get close to peak performance, provided program optimizations take into account the detailed workings of each processor component, as well as the interactions among components,
- the architecture components which have the greatest impact on performance, and the program optimizations which bring the biggest performance improvements are not necessarily the ones on which most research works focus,
- executing even a simple program on a modern processor results in complex behaviors, but that understanding these phenomena is both feasible and critical to design relevant and efficient compiler optimizations in the future.

To illustrate these different points, we needed to select a program optimization domain (memory optimizations) and a target code (a simple and classic regular numerical kernel like

0-7695-1524-X/02 $17.00 © 2002 IEEE

Matrix-Multiply) on which *lots* of research efforts have already been spent [2],[15],[25],... Since the purpose of this research work is *not* to design new program optimization techniques but to understand whether it is possible and what it takes to reach high sustained performance, we have collected the major and most relevant program optimization techniques in the literature, those performed in the Alpha production compiler, the associated KAP preprocessor [8], and those used in the Alpha EV6 libraries. Using a subset of these transformations: (1) we outline a sequence of transformations that achieves a 3.65-fold improvement over the kernel optimized with the Alpha compiler, a 3.52-fold improvement over traditional cache optimizations using tiling [10], and 95% of the Alpha peak performance; (2) we use this sequence of optimizations to precisely quantify and analyze the performance improvement brought by each architecture component/program optimization pair.

We wish to point out that we are fully aware that Matrix-Multiply is *not representative* of regular programs [9], but this is less a concern in the present study since we wish to focus on and highlight the *interaction* between program and architecture rather than design program transformation techniques that will apply to a large array of programs. While we cannot draw general conclusions from the analysis of a single kernel, this analysis certainly suggests that the research effort on program optimizations should be much more architecture-aware if the performance gap between peak and sustained performance is to be bridged.

Our approach differs from classic compiler optimization research in that we are strongly tied to the architecture, all optimizations are driven by simulator-based *dynamic* analysis in opposition to compile-time *static* analysis, and the impact of each transformation on all architecture components is carefully evaluated. For this study, we had access to the full EV6 processor simulator from Compaq so that we could analyze in details the impact of transformations on all architecture components. Even though all experiments are conducted on the Alpha, the findings and conclusions of this study are very likely to be relevant to other high-performance processors.

In Section II we briefly present the target processor architecture and the experimental framework; Section III is the core of the article where we present the detailed architecture-oriented analysis and optimization of the Matrix-Multiply kernel and the impact of each architecture component on performance; finally, in Section IV we outline a first sketch of a practical approach to program optimization, intermediate between fully manual optimization and fully automatic optimization, which consists in formalizing our experience in program optimizations in such a way that it can be delivered and exploited by performance programmers.

II. Experimental Framework

The target architecture is an Alpha 21264 processor (EV68) clocked at 1GHz. The architecture is described in Figure 1, where Ebox (integer) and Fbox (floating-point) respectively contain 4 and 2 functional units, and Ibox is the instruction fetch mechanism capable of fetching 4 instructions and issuing 6 in the same cycle. The ICache and DCache are 64 KB 2-way set-associative caches with 64-byte blocks. The BCache is a 8MB direct-mapped cache with 64-byte blocks. The Memory reference unit (Mbox) controls the DCache and ensures architecturally correct behavior for load and store instructions. The Mbox contains a Load Queue and a Store Queue of 32-entry each, a Miss Address File (MAF) to coalesce pending misses on the same cache block, and the TLB which holds 128 entries and is fully-associative; page size is 8 KB. The register bank contains 80 integer and 72 floating-point registers.

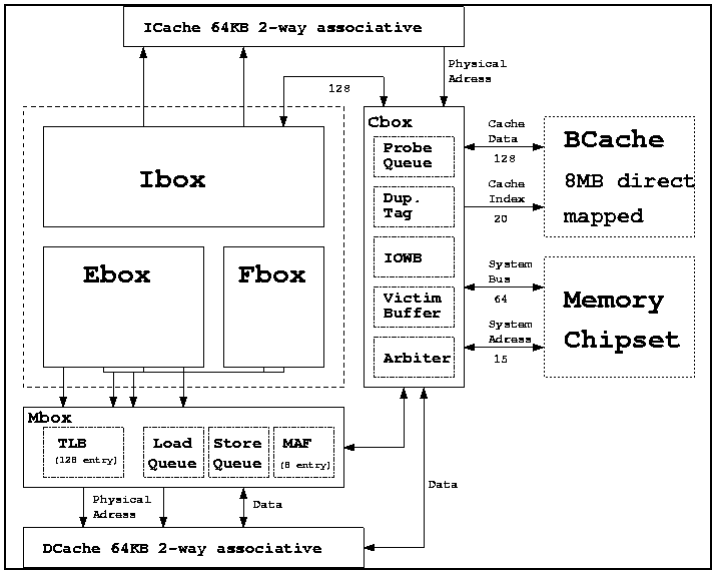

Figure 1. EV6/Alpha 21264 processor architecture

The experiments of this study were conducted for a large set of matrix dimensions. In Table 2, we report experimental values with dimension N≈1000 (exactly, execution times are average of the execution time of several matrix dimensions in an interval centered on N=1060); N≈1000 realizes a reasonable tradeoff between having large matrices and running processor simulations in a reasonable time.

III. An Architecture-Aware Analysis and Optimization of Matrix-Multiply

```
do i=1, Ni
   do k=1, Nk
    R=b(k,i)
      do j=1, Nj
         c(j,i)=c(j,i)+R*a(j,k)
      enddo
   enddo
enddo
```

Figure 2. Original Matrix Multiply program C=A×B

As mentioned in the introduction, we attempt to squeeze as much performance as possible out of the Matrix-Multiply kernel by carefully considering each component of the architecture, one after the other. For each architecture component, we used the dynamic analysis provided by

simulations to understand which phenomenon occurs on this component, and each time we tested a variety of transformations and picked the best performer. As a result, we have obtained a sequence of transformations that isolates and highlights the impact of each architecture component on overall performance, and the contribution of each program transformation. Finding the best *sequence* of transformations required a long trial and error process coupled with detailed dynamic analysis. The list of transformations is indicated in Table 1 along with the architecture components targeted. In Table 2, the performance of different program versions is indicated, along with the corresponding sequence of transformations used in each version. The base version, i.e., Step00, is compiled with `-O2 –unroll 1 –nopipeline`[1] so as to perform most classic compiler optimizations (common subexpression elimination, strength reduction, code scheduling, inlining,…), but not the more architecture-oriented optimizations found in `–O4` and `–O5`, especially loop unrolling, software pipelining and prefetching.

As can be seen in Table 2, the last version achieves a 13.56 speedup over the base version and 95% of the peak performance, a 3.65 speedup over the `-O5` version (best compiler optimization), and a 3.52 speedup over the `–O5` version augmented with the KAP preprocessor [8]. The overall speedup is similar to the speedup of the library [26][27] which is another example of manual best effort. For this study, we have analyzed in details the different optimizations proposed in the literature, performed in the compiler, the preprocessor KAP and the library, and using a processor simulator, we have identified how these optimizations operate on architecture components, and we have assembled a set of optimizations from these different sources. Through the analysis of each architecture component, we show how program optimizations should be defined and driven by the program behavior on the architecture, which architecture component has the greatest impact on performance and how different components can interact in complex ways,

In the paragraphs below, we consider each architecture component in turn. At the beginning of each paragraph, we outline the speedup achieved with the corresponding program optimization, and the contribution to the final speedup. Note that we focus on the contribution to the speedup rather than the contribution to the execution time variation, because the order of the sequence of optimizations matters.

[1] « -unroll 1 » inhibits unrolling, while « -nopipeline » inhibits software pipelining.

Step	List of Transformations
Step00	Original program
Step01	Blocking 2D for DCache
Step02	Blocking 3D for TLB
Step03	Blocking 3D + Loop interchange for Store Queue
Step04	Blocking 3D + Loop interchange + Unrolling for ILP
Step05	Blocking 3D + Loop interchange + Unrolling + Blocking for Registers
Step06	Blocking 2D for DCache + Copying for TLB + Loop interchange + Unrolling + Blocking for Registers
Step07	Blocking 2D for DCache + Copying for TLB + Loop interchange + Unrolling + Blocking for Registers + Prefetch
Step08	Blocking 2D for DCache + Copying for TLB + Loop interchange + Unrolling + Blocking for Registers + Prefetch + Blocking 3D for BCache
Step09	Blocking 2D for DCache + Copying for TLB + Loop interchange + Unrolling + Blocking for Registers + Prefetch + Blocking 3D for BCache + Optimizations for architecture-specific features

Table 1. Program versions

Step	Speed-Up
Step00	1.00
Step01	2.48
Step02	2.62
Step03	3.11
Step04	3.71
Step05	9.90
Step06	8.43
Step07	12.25
Step08	12.75
Step09	**13.56**
`-O5`	3.26
`-O5 with KAP`	3.37

Table 2. Speed-ups for N ≈ 1000

L1 CACHE (*Step01*, speedup=2.48, contribution=12%). Most of the research works on regular numerical codes, and especially on Matrix-Multiply, focus on improving cache behavior because of the high miss ratio induced by intensive memory usage : 32% for the DCache, i.e., the EV6 L1 cache.

```
Cache tiling  →   do kk=1,Nk,T
Cache tiling  →     do jj=1,Nj,T
                      do i=1, Ni
                        do k=kk,MIN(kk+T-1,Nk)
                          R=b(k,i)
                            do j=jj,MIN(jj+T-1,Nj)
                              c(j,i)=c(j,i)+R*a(j,k)
```

Figure 3. Step01: *2D tiling for caches*

The most popular optimization method for programs with significant and poorly exploited temporal locality like Matrix-Multiply is tiling [18][2][10]. The block size of matrix *A* must be carefully picked to minimize conflict misses [2]; it is interesting to note that block size selection algorithms which solely rely on miss cost functions [10] perform significantly worse than algorithms which rely on global execution time

[19], and which implicitly consider all architecture components. In our case, we performed an exhaustive search to find the block size value that minimizes execution time and we found *T=33*.

Even though the optimization results in dramatic miss ratio reduction on the DCache (95%), it only accounts for 12% of the final speedup.

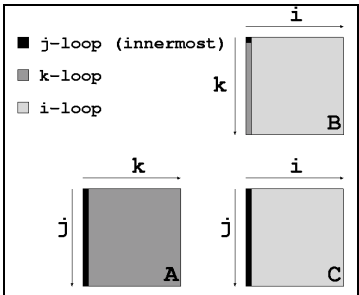

Figure 4. Before cache tiling

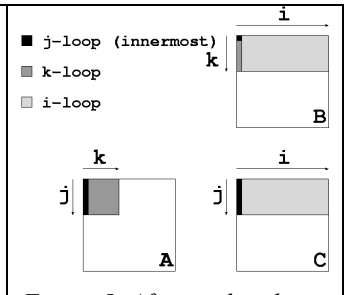

Figure 5. After cache tiling

L1 CACHE + TLB (*Step02*, speedup=2.62, contribution=1%). Only recently [5],[10], researchers have pointed out that cache tiling can have adverse effects on the TLB. Consider Figure 4 and Figure 5, matrices *B* and *C*, and assume one matrix column is roughly the size of one page (8 kB in the Alpha EV6); in the non-tiled version, on each iteration of *i,* a column of *B* and *C* is accessed and the corresponding TLB entries are loaded. These TLB entries are respectively reused on all iterations of the *k* loop (for matrix *B*) and *j* loop (for matrix *C*), and the reuse distance is respectively one iteration of the *k* and *j* loops. Now, in the tiled version, only part of the column of each matrix is accessed in loops *k* and *j*; therefore the TLB entry of matrix *B* is further reused on all iterations of loop *kk*, and the TLB entry of matrix *C* on all iterations of loop *jj*. Consequently, the reuse *distance* varies: for the *k* and *j* loops it is still one iteration of the inner loop, but for the *kk* and *jj* loop it is one *execution* of the whole *i* loop nest. As a result, the reuse distance is very large and the probability the TLB entry is flushed before it can be reused is significantly increased, resulting in additional TLB misses in the tiled loop: the DTLB miss ratio is increased to 5.08e-4 versus 4.98e-4 in the original loop (TLB miss ratios are fairly low, but the cost of one TLB miss is significantly higher than the cost of a cache miss).

```
TLB tiling  →   do ii=1,Ni,T
                  do kk=1,Nk,T
                    do jj=1,Nj,T
                      do i=ii,MIN(ii+T-1,Ni)
                        do k=kk,MIN(kk+T-1,Nk)
                          R=b(k,i)
                            do j=jj,MIN(jj+T-1,Nj)
                              c(j,i)=c(j,i)+R*a(j,k)
```

Figure 6. Step02: *3D Tiling for TLB*

We do not want to loose the benefit of cache tiling, but we want to minimize its impact on the TLB. For that purpose, we want to manage the reuse of TLB entries on the *i* loop much the same way we managed cache data reuse on the *j* and *k* loops: we can *tile* the *i* loop so that the number of TLB entries of matrices *B* and *C* in a tile is small enough to fit in the TLB, and can be reused on the *kk* and *jj* loops. The TLB miss ratio is reduced from 5.08e-4 to 2.08e-5. The number of cache misses is slightly increased but not enough to compensate for the benefit of the TLB miss reduction. The speedup increases to 2.62.

L1 CACHE + TLB + STORE QUEUE (*Step03*, speedup=3.11, contribution=4%). Stores are usually not considered as prime candidates for optimizations because stores are typically not critical operations: data is sent back to memory and will not be used shortly. However, several architecture components can degrade overall processor performance when stores behave poorly, e.g., store queue, write buffer or victim address file. For instance, when the write buffer is full, any additional write will stall the cache and possibly the processor, degrading overall performance; it is not the case in our example.

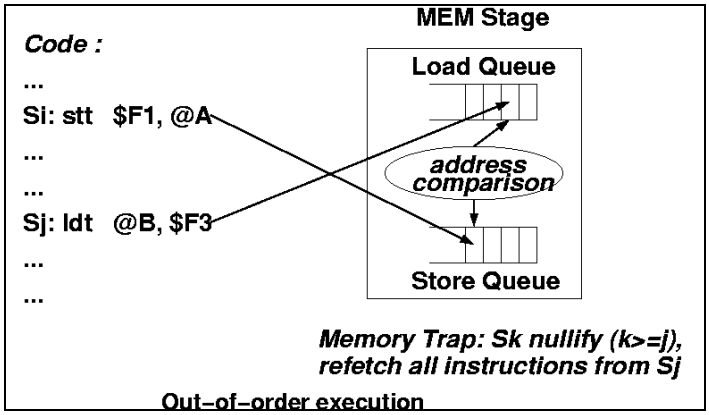

Figure 7. Store Queue and load speculation

In superscalar processors, there are some cases where memory operations must be aborted; the instruction is then fetched again and executed; in the EV68 such cases are called *replay traps*. In the EV68, two memory operations cannot be in progress at the same time, see Figure 7, if they map to the same cache set, due to architectural limitations. If that happens the newest instruction is aborted and a replay trap occurs.

In version *Step02*, the number of replay traps is fairly high because the innermost loop nest contains one store and two loads which sometimes access the same cache set. One way to reduce the probability that conflicts, and thus replay traps, occur is to reduce the number of load or store operations. For that purpose, we interchange inner loops *k* and *j*, see Figure 8. Then, the write request *c(j,i)* can be moved out of the inner loop, dividing by *T* (=33) the number of store requests. As a result, the number of replay traps is divided by 24, and the speedup increases to 3.11.

```
                        do ii=1,Ni,T
                          do kk=1,Nk,T
                            do jj=1,Nj,T
                              do i=ii,MIN(ii+T-1,Ni)
Loop interchange →              do j=jj,MIN(jj+T-1,Nj)
                                  R=c(j,i)
Loop interchange →                do k=kk,MIN(kk+T-1,Nk)
                                    R=R+b(k,i)*a(j,k)
                                  Enddo
                                  c(j,i)=R
```

Figure 8. Step03: *Loop interchange for Store Queue*

L1 CACHE + TLB + STORE QUEUE + ILP (*Step04*, speedup=3.71, contribution=5%). Registers are a critical resource and large loop bodies usually result in spill code rather than unexploited registers. But in a small loop like Matrix-Multiply, there are multiple available registers. Compilers usually employ additional available registers to increase instruction-level parallelism using loop unrolling and software pipelining [1]. Similarly, we unroll the innermost loop 8 times (which we denote 118 unrolling: unroll 1 on *i*, 1 on *j* and 8 on *k*), and the speedup is now 3.71.

L1 CACHE + TLB + STORE QUEUE + ILP + REGISTERS (*Step05*, speedup=9.90, contribution=49%). At this point, even though we have significantly reduced the number and effect of cache and TLB misses, i.e., we have reduced the average memory latency, our program is still *memory bound* because the ratio of the number of computations over memory accesses (load/store) is low (≈1). However, registers, the uppest level in the memory hierarchy, can also be used to reduce the number of memory accesses. And after the ILP optimization of the previous paragraph, we found that several registers are still available. So, we can now use these additional available registers to reduce the number of load/stores by treating registers as *just another memory hierarchy level*. We *tile* for the register level simply by *unrolling outside loops* [23]. Consider the program of Figure 8 and the same program below where the *j* loop has been unrolled twice.

```
                        do ii=1,Ni,Ti
                          do kk=1,Nk,Tk
                            do jj=1,Nj,Tj
                              do i=ii,MIN(ii+Ti-1,Ni)
Loop unrolling →                do j=jj,MIN(jj+Tj-1,Nj),2
                                  Rc0  = c(j,i)
                                  Rc1  = c(j+1,i)
                                  Rb0 = b(k,i)
                                  do k=kk,MIN(kk+Tk-1,Nk)
Register reuse →                    Rc0 = Rc0 + Rb0 * a(j,k)
                                    Rc1 = Rc1 + Rb0 * a(j+1,k)
```

Figure 9. Step05: *Register tiling*

The *b(k,i)* reference can be placed in a register, avoiding a memory access; similarly, if we unroll *j* U times we would avoid U-1 memory accesses. We can do the same with loop *i* and reference *a(j,k)*. We experimentally found that optimal performance is achieved for several different combinations of unroll factors, and we select 348 for now (we will later change again the unroll factor). We observe that the number of loads decreases by 69%, the instruction-level parallelism is as high as for innermost loop unrolling (118, see previous paragraph), and the ratio of computations over memory operations is increased by 300%. Consequently, functional unit utilization is higher and overall performance is increased by 49%, with a speedup of 9.90.

Therefore, by treating registers as an additional memory level, we can tremendously improve the program *bandwidth*; while it may be difficult to exploit physical registers in current superscalar architectures because they are hidden by dynamic renaming [12], new instruction set architectures like the Itanium EPIC [13] expose much more logical registers to the programmer and the compiler.

EXAMPLE OF INTERACTION BETWEEN L1 CACHE, TLB, STORE QUEUE AND REGISTERS (*Step06*). The strong speedup brought by register tiling suggests we should probably privilege this optimization over other optimizations. Therefore, in this paragraph, we revisit several other optimizations in order to maximize register tiling, in order to illustrate how interactions among architecture components can influence individual optimizations and the sequence of optimizations.

Note that in the above paragraph, we have improved the register utilization for reference *a(j,k)* and reference *b(k,i*), but we have not mentioned reference *c(j,i)*. Loop unrolling does not improve register utilization for reference *c(j,i)*, it is the size of loop *k* (*Tk*) that determines register utilization for this reference: the longer loop *k*, the fewer the number of load/stores. Since we use a relatively small block size (*Tk*=33), we could achieve significant performance improvements with a much larger *Tk* tile size.

```
                        do i=1,Ni,Ti
                          do kk=1,Nk,Tk
                            do j=1,Nj,2
                              Rc0  = c(j,i)
                              Rc1  = c(j+1,i)
                              Rb0 = b(k,i)
                              do k=kk,MIN(kk+Tk-1,Nk)
Register reused →               Rc0 = Rc0 + Rb0 * a(j,k)
Register reused →               Rc1 = Rc1 + Rb0 * a(j+1,k)
```

Figure 10. Trading L1 reuse for register utilization

If we increase *Tk* to improve register utilization, we will not be able to still exploit the L1 reuse of matrix *A*, because its *Tj*x*Tk* tile would not fit in cache anymore. We have made several tests which showed that, in the end, it is worth dropping the L1 reuse of matrix *A* to favor the register utilization of matrix *C* and the subsequently enabled optimizations described in the next sections, like prefetching.

However, while we loose the L1 reuse of matrix *A* we can have a large *Tk* value and still exploit the L1 reuse of matrix *B*, which partly compensates for the loss of matrix *A* L1 reuse. Indeed, we have seen that loop *i* is unrolled 3 times for register utilization purposes (unrolling factor is 348), so the *B* tile reused on each iteration of loop *j* is a 3x*Tk* tile. Therefore,

we need to find the biggest possible value of *Tk* such that the 3x*Tk* tile fits in the L1 cache. Because the tile is very flat, we can fulfill our goal of having a large value of *Tk*; in fact, we find *Tk=Nk* satisfies this constraint. The corresponding code is shown in Figure 10. Note that loop *j* is not tiled anymore as we have given up on the L1 reuse of matrix *A*, and similarly, we have dropped the associated TLB tiling of loop *i*.

```
Copy →      do kk=1,Nk,Tk
               do i=1,Ni, 3
            C     COPY of matrix B
Copy →                …
               do j=1, Nj, 4
            C     COPY of matrix A
                  …
                  Rc0  = c(j,i)
                  …
                  do k=kk,MIN(kk+Tk-1,Nk), 8
            C       Innermost loop
                  end do
                  c(j,i) = Rc0
                  …
               end do
            end do
         end do
```

Figure 11. Copying to avoid TLB misses

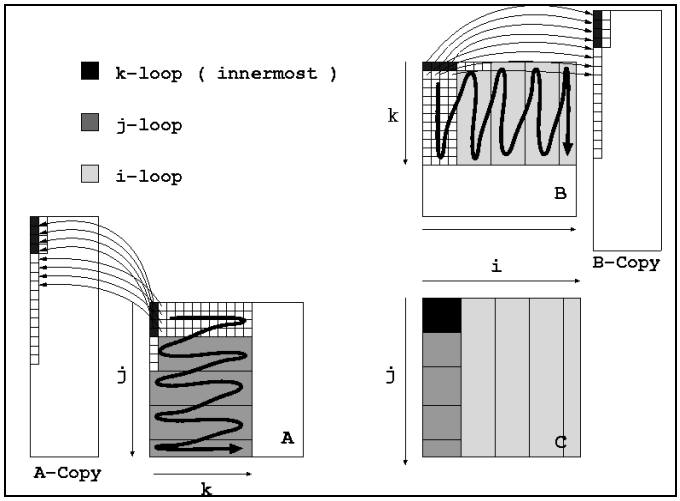

Figure 12. New tiling pattern

Now, if we tile with these values, we achieve a slowdown of 48% because the number of TLB misses has been multiplied by 780. The value of *Tk* is such that the TLB is trashed *within the innermost loop k* because of reference *a(j,k)*, and to a much lesser extent, reference *b(k,i)*. TLB tiling on loop *i*, as used in *Step02*, has the effect of reducing TLB misses *across executions* of loops *j* and *k*, but it has no effect on TLB misses *within* loop *k*. We can avoid trashing the TLB by *copying* the *A* tiles so they become contiguous memory addresses, see Figure 12 and the code in Figure 11; because the *A* tile is reused several times, the benefit of copying compensates for the copying overhead cost in terms of TLB misses. Moreover, copying has the additional benefit of reducing L1 cache conflict misses, see [14]. Using copying, the number of TLB misses is reduced by 98%.

Finally, in a previous paragraph, we had unrolled the inner loop 8 times (118), and based on this constraint we had unrolled outer loops as much as register usage allowed. Implicitly, we had privileged instruction-level parallelism over register tiling. Now, we have observed that register tiling bring significant benefits, so we reverse this trend, and privilege register tiling over instruction-level parallelism and the best unrolling factor that satisfies this new constraint is 442.

However, the overall speedup brought by this version is only 8.43 (*Step06*) versus 9.90 (*Step05*) in the previous version, essentially because caches are still rather poorly exploited (especially matrices *A* and *C*) since we have focused on registers, and because of the overhead of copying. However, we will see in the next paragraph, that the current version is now better suited to the next optimizations, especially prefetching, and will achieve a higher overall performance improvement; a similar register-oriented approach is used in the Alpha library [27].

L1 CACHE + TLB + STORE QUEUE + ILP + REGISTERS + PREFETCH (*Step07*, speedup=12.25, contribution=19%). If we directly apply prefetching to the initial Matrix-Multiply version *Step00* (for all matrices), we obtain a rather small performance improvement (speedup=1.33 over *Step00*) because poor cache and register utilization results in high memory traffic, and almost no memory bandwidth available for prefetching. If we apply prefetching to our most efficient version until now, i.e., *Step05*, we only achieve a speedup of 1.05 over *Step05* because of the small square tiles (*T*=33): there are 8 cache lines per tile (one cache line is 64 bytes), so that there is at most 4 prefetching opportunities within one tile, and in fact even less, since we need to use a large enough prefetch distance to compensate for the memory latency.

Now, in the Matrix-Multiply version of the previous paragraph (*Step06*, a slowdown of 0.90 over *Step05*), we use much larger tiles (*Tk=Nk*) which are better suited to prefetching. Besides, register tiling and TLB miss reduction help keep memory traffic reasonably low. Finally, besides reducing the average memory latency, prefetching would bring the added benefit of hiding the overhead of copying in this Matrix-Multiply version.

```
                  do kk=1,Nk,Tk
                     do i=1,Ni,4
                  C     COPY of matrix B
Prefetching →           Prefetch( b(k+4,i) )
                     …
                     do j=1,Nj,4
                  C     COPY of matrix A
Prefetching →           Prefetch( a(j,k+4) )
                     …
                     Rc0=c(j,i)
                     …
                     do k=kk,MIN(kk+Tk-1,Nk),2
Prefetching →           Prefetch( a(j,k+4) )
                        Rc0=Rc0+b(k,i)*a(j,k)
                        …
```

Figure 13. Step07: *Prefetching*

In this version, we can apply prefetching in several places: when copying the tiles of matrices *A* and *B* to hide BCache to memory latency, and when fetching matrix *A* in the innermost loop *k* to hide DCache to BCache latency, since the DCache (L1) reuse of matrix *A* has been dropped, as mentioned in the previous paragraph. If it were possible to prefetch at the source-code level, the corresponding code would look like Figure 13;, however, in the Alpha EV6, prefetching support is only available at the assembly-code level, though normal loads to the null register $31, as follows

```
ldl   $31, 16*8($5);    prefetch a(j,k)
```

The prefetch distance, i.e., the number of iterations between the array element being currently used and the array element being prefetched, is a delicate tradeoff: if it is too large, too many array elements will not benefit from prefetching, especially in a tiled loop, if it is too small compared to the memory or cache latency, the data does not arrive soon enough to avoid a stall [33]. There is no such issue for outer loop prefetching (prefetching applied to copying) because data is not used immediately after it is fetched, but the prefetch distance of the innermost loop (fetching matrix *A* in loop *k*) needs to be finely tuned; in that case, we found that a prefetch distance of 2 iterations realizes the best tradeoff.

When prefetching is used the speedup over *Step00* is now 12.25, i.e., higher than if prefetching is applied to *Step05* (10.37).

L1 CACHE + TLB + STORE QUEUE + ILP + REGISTERS + PREFETCH + L2 CACHE (*Step08*, speedup=12.75, contribution=4%). Tiling algorithms usually focus on L1 caches. However, lower-level caches usually have a much larger latency: for our EV6 system, the miss latency of the DCache (L1) to the BCache (L2) is 12 cycles, while the miss latency of the BCache to memory is 140 cycles. Moreover, recent research works [28] have shown that a significant share of L1 miss latency is hidden by the dynamic reordering of instructions in out-of-order processors like the EV6, which means reducing the number of L1 misses may not always result in significant performance improvements. On the other hand, optimizing for a cache hierarchy instead of a single cache level forces to realize a difficult tradeoff between the different cache levels [15],[16]: tiling for one level can increase conflict misses in another level or can reduce the amount of achieved reuse.

In the current program version, we use 4x*Tk* tiles of matrix *B* for the L1 cache (because the unrolling factor is 442), and these tiles are fetched following the *k* dimension first, and then following the *i* dimension, see Figure 12. Implicitly, it means the 4x*Tk* tile of matrix *B* (see previous paragraph) is fully used, i.e., against all corresponding 4x*Tj* tiles of matrix *A* (recall loop *j* is unrolled 4 times also), before it is discarded. On the other hand, each tile of matrix *A* is reused on each iteration of loop *i*. Consequently, the reuse distance, i.e., one *execution* of loop *j,* is fairly large so that even an L2 cache may not exploit this reuse. To reduce this reuse distance, we tile the *i* loop, and we then need to tile the *j* loop as well otherwise the reuse distance for matrix *B* would now become too large. The resulting code is much like the 2D and 3D tiling used in previous paragraphs for L1 cache and TLB, but this time, it targets the BCache (L2 cache) and has the effect of keeping tiles of matrices *A* and *B* in the L2 cache. The additional tile levels and the corresponding tile traversal order are shown in Figure 14 and Figure 15 below.

```
L2 Tiling →      do ii=1,Ni,Ti
                   do kk=1,Nk,Tk
                 Outer Copy
L2 Tiling →        do jj=1,Nj,Tj
                 Inner Copy
                     do i=ii,MIN(ii+Ti-1,Ni),4
                       do j=jj,MIN(jj+Tj-1,Nj),4
                         Rc0  = c(j,i)
                         …
                         do k=kk,MIN(kk+Tk-1,Nk), 2
                 C                Innermost loop
                                  …
```

Figure 14. Step08: *L2 tiling*.

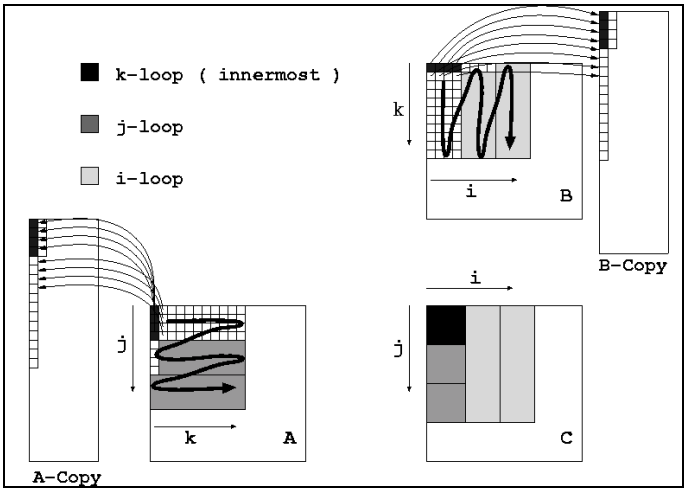

Figure 15. Step08: *New execution pattern.*

In our EV6 workstation, the BCache has a size of 8 Mbytes, so that for *Ni=Nj=Nk*=1060, the matrices do not fit in the BCache. The experiments show that the number of BCache misses is reduced by 11%, and the speedup increases to 12.75. Note that this additional tiling level disrupts DCache reuse as expected and DCache miss ratio increases from 9% to 10%. However, the much larger latency of BCache misses more than compensates for the increase in DCache misses.

L1 CACHE + TLB + STORE QUEUE + L2 CACHE + PREFETCH + REGISTERS + ILP + ARCHITECTURE-SPECIFIC FEATURES (*Step09*, speedup=13.56, contribution=7%). Most of the abovementioned components, e.g., cache hierarchy, TLB, registers, are "classic" components in the sense that their presence in an architecture is well-known even though program optimizations do not always target them all, or at least not simultaneously. The store queue usually attracts less attention though it is now a component of most superscalar processors so we decided not to include it in the "architecture-specific feature" category. However, besides these "mainstream" components, a processor architecture is full of specific features which reflect the tradeoffs made by computer architects. Sometimes, such features can have a significant

impact on performance and can play a role in bridging the gap between peak and sustained performance. As in all processors, there are many such features in our target architecture, and the purpose of this paragraph is to illustrate the impact of such features with one of the EV6 features.

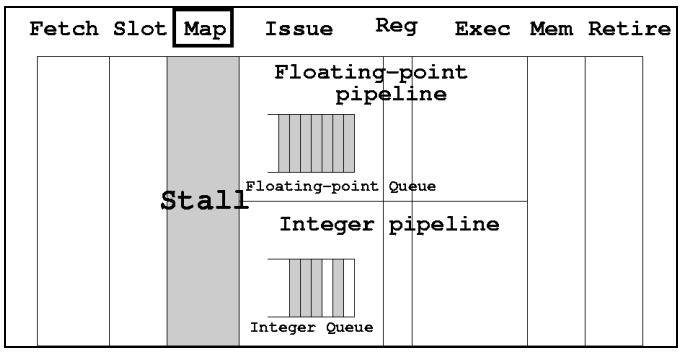

Figure 16 : Floating-point queue stall can stall

In the Alpha EV6, there are two pipelines: an integer pipeline and a floating-point pipeline; both pipelines feed their respective functional units, and buffer instructions in *queues*: an *integer queue* and a *floating-point queue*, see Figure 16. When the integer and the floating-point queues contain many instructions, all physical registers can be in use, so that when a new instruction is fetched, it cannot be allocated a physical register in the *Map* pipeline stage (the role of the *Map* stage is to map and translate the logical registers of assembly instructions into physical registers). As a result, the *Map* stage will stall. However, in the Alpha EV6, when the *Map* stage stalls, it cannot start again immediately after physical registers were freed, restarting requires two additional stall cycles, so that stalling the *Map* stage specifically is fairly costly in this architecture. As mentioned above, this situation can occur when floating-point and integer queues tend to fill up because instructions are fetched at a higher rate than they can be processed; it happens most often with the floating-point queue which has long-latency instructions, and it is the case in our Matrix-Multiply program. Now, if we can *slow down* the flow of floating-point instructions, we would avoid filling up the floating-point queue and then stalling the *Map* stage, which would increase overall performance without degrading the processing rate of floating-point instructions. One solution is to introduce NOPs in the flow of instructions: a NOP goes in the integer queue which is barely used in Matrix-Multiply, slows down the fetch rate of floating-point instructions and avoids stalling part of the processor.

These NOP instructions are inserted at the assembly level, like prefetch instructions, as shown below.

```
    add   $f18, $f12, $f18          add   $f18, $f12, $f18
                                    nop
    mul   $f14, $f26, $f12          mul   $f14, $f26, $f12
    ld    $f8,  0*SIZE($3)          ld    $f8,  0*SIZE($3)
                                    add   $f17, $f11, $f17
                                    nop
    mul   $f14, $f27, $f11
    ld    $f19, 0*SIZE($5)
```

Figure 17. Step09: *Inserting NOPs*.

Using this NOP-augmented routine, we find that floating-point queue stalls decrease by 60%, and overall performance is increased by 7%, and the speedup is now 13.56 over *Step00*.

SUMMARY In the end, the total speedup is equal to 13.56 with respect to the base version, and corresponds to 95% of the peak performance, where the peak performance is the maximum number of floating-point instructions per second, i.e., 2 GigaFlops.

IV. TOWARD A PROGRAM OPTIMIZATION STRATEGY

Examples like Matrix Multiply highlight the need for a much more architecture-oriented approach to program optimization. Though we hope that compilers will ultimately prove capable of harnessing processor architecture complexity, in the meantime, we need to find a practical approach for addressing the widening gap between available and achieved performance. The issue essentially rests with performance-oriented end-users who wish to extract the best possible performance out of their workstations and servers, and with computer manufacturers and vendors who need to demonstrate the actual performance of their products to would-be customers; note also that with the increased popularity of *metacomputing*, an execution time reduction of R% on the single-processor performance of a given program, directly translates in R% less computers needed to perform the same task, so even a small performance reduction implies significant economical gains in that context.

However, we cannot expect end-users and even most engineers at computer manufacturers to be proficient in processor architecture, especially considering the current processor architecture complexity. Conversely, we do not believe a fully-automatic, compiler-like, solution is a reasonable short-term expectation.

Consequently, we are currently investigating a short-term practical approach which is intermediate between fully manual optimization and fully automatic optimization. This approach relies on *dynamic* analysis, which has the tremendous advantage, over the *static* analysis used in compilers, to unveil the *precise* program behavior on the architecture. The principle of our approach is to *formalize* our experience in program optimizations in such a way that it can be delivered and exploited by engineers who are reasonably aware of processor architectures but not experts on the matter. This "experience" consists in organizing program optimization in a sequence of guided steps, i.e., an *optimization strategy*. Surprisingly enough, there is a huge amount of research work

on program transformations for local program structures and specific architecture components, but almost no research work on how to organize the problem of optimizing a *whole* program on a *whole* architecture.

We are currently working on delivering our experience on program optimization in the form of a *decision tree*, which consists in a set of *analysis, decision, and optimization steps*; Figure 18 illustrates this notion with a very simple example. At each step, the tree guides the end-user by suggesting an analysis or an optimization based on previous analysis results. The optimization strategy iterates over four steps:

1. At each step, determine the current target architecture component; at this point the analysis is rather coarse, e.g., number of L1 cache misses.
2. Then determine if the problem is *local* to several program structures, i.e., several loop nests which breed most misses, or *distributed* in the program, e.g., misses are inter-nest misses [9] because data are not reused from one loop nest to another. In the latter case, program transformations must be applied to make the problem more "local", e.g., for instance we can merge loop nests (or, depending on the problem, inline procedures, modify data structures...).
3. Once the problem has been decomposed into a set of "local" problems, or if we find it is already local in step 2, we can start the detailed architecture-oriented program analysis to identify the nature of the problem, much like we did for Matrix Multiply, and then suggest appropriate transformations.
4. Once the behaviour of this architecture component has been improved we can iterate and go back to step 1.

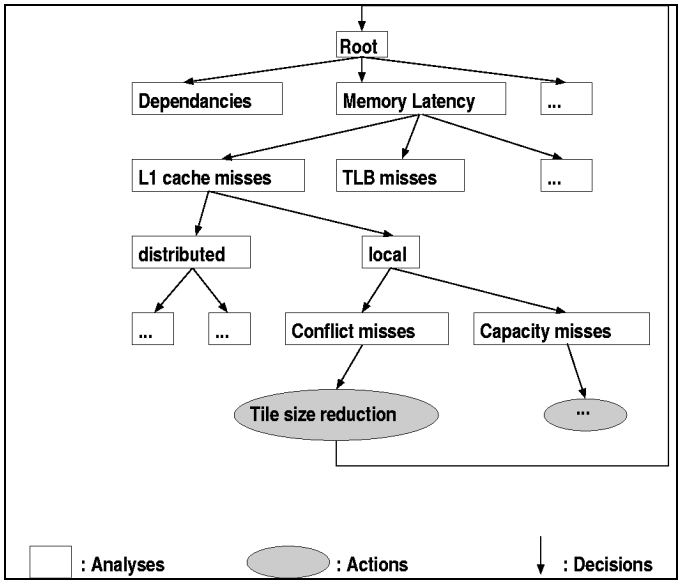

Figure 18. Notion of Decision Tree

At the top of the decision tree, we find all possible *performance anomalies* of the architecture, i.e., all possible sources of performance degradations. For instance, some of the root anomalies are instruction dependencies, functional units not used, large average memory latency, sub-optimal instruction fetch, processor traps… In subtrees, we find in-depth analysis of each component. For instance, instruction dependencies are split into integer and floating-point dependencies, which are themselves split into low-latency fully-pipelined floating-point operations (additions) and high-latency partially-pipelined floating-point operations (divisions), and so on… This level of detail enables precise user guidance during the optimization process.

V. Future and On-Going Work

This whole optimization strategy raises several practical issues:

- How do we build such a decision tree ?
- How do we perform dynamic analysis in practice, i.e., rapidly obtaining detailed information on the program behavior on the architecture ?
- How do we apply program transformations in practice ?
- When do we stop the iterative optimization process ?
- What is the impact of the data set on the sequence of optimizations ?

While a detailed explanation of the optimization strategy is out of the scope of this article and will be published in another article, we provide several hints at the answers to these main questions.

Decision Tree. Decision trees are built *empirically* using manual optimizations of a set of programs. Currently, we are using the SpecFP2000 benchmarks to build a decision tree for this type of programs; ultimately, we may have one decision tree per program "corpus", e.g., one for floating-point codes and one for integer codes, and possibly more refined distinctions, like finite-difference solvers, finite-element solvers, direct solvers, etc.

In order to build a decision tree, we manually optimize each program, then we backtrack and consider the sequence of analysis/decision/optimization steps used, and we insert the corresponding branches in the decision tree. As we consider more programs, we better understand in *which order* architecture components need to be considered and which optimizations need to be applied; for instance, in the Matrix Multiply kernel, it is useless to optimize for the TLB before the caches are considered, or to optimize for the L1 or L2 cache before the registers are considered, etc. Besides, as we consider more programs, fewer branches are added and the tree becomes more stable. Once the tree is sufficiently stable, it can be used for optimizing other programs.

Dynamic analysis. Using dynamic analysis means using a processor simulator. The main flaw of processor simulator is their excruciatingly low speed: a simulated program runs several thousand times slower than the original program on the same machine. In order to make dynamic analysis a practical tool, we use statistical simulation [29] and fast simulation techniques (not yet published), so that we can reduce the slowdown to an acceptable 50 to 100 compared with 3000 with the public *SimpleScalar* [17] simulator for instance.

Applying program transformations. Some program transformations are difficult to apply for a non-expert user because they require legacy checking, e.g., dependence checking. Moreover, there is a large collection of such transformations. For that purpose, we will attempt to unify as much as possible the different transformations into a few compound transformations, delivered through a transformation tool such as SUIF [30], in order to simplify the task of applying transformations. Parameter tuning of program transformations, e.g., tile size, is the most a delicate task [2], but recently proposed techniques, e.g., based on genetic algorithms [19], show it is possible to automate this part of the optimization process.

Stopping the iterative process. One of the critical issues is to decide *when* we can stop the iterative optimization process. Peak performance provides a first metric, but in many cases, it is far too raw: consider for instance the Matrix-Multiply kernel and assume we replace floating-point multiplications and additions by floating-point divisions. Then, we know the peak performance of this program is not 2 GigaFlops but 0.66 GigaFlops, since the Alpha EV68 can issue at most 1 floating-point division every 15 cycles. Therefore, we are also working on defining more precisely the notion of *optimal performance* for a given program. For instance, we have developed a practical approach to the problem of finding what would be the program execution time if all cache and TLB misses were removed [21], i.e., a performance upper-bound for memory latency issues.

Sensitivity to data set. To build the data tree, we use a single data set per program. Different data sets can naturally induce different program behaviors; fortunately, the fact we use a set of programs instead of a single program partially compensates for this weakness. However, for other aspects of our methodology, like tuning program optimization parameters, we need to better understand the sensitivity of program behavior to data sets. Initial work on prefetch distance again shows this task is tractable [20] but needs to be further pursued, and recent results in iterative compilation [31],[11] will provide a useful feedback.

CONCLUSIONS

In this article, we have shown that the behavior of even simple programs on architecture is very complex because multiple architecture components interact at the same time. However, we have also shown that it is possible to harness this complexity and deduce from detailed dynamic analysis how the program must be modified to improve its behavior on architecture. We have also shown that cache tiling, on which a large share of research works focus, only accounts for 12% of the total performance improvement, suggesting research works not always investigate the most critical architecture components or the most appropriate program optimizations.
This simple example suggests that program optimizations need to be revisited to take into account rapidly evolving processor architectures, and until this progress can be reflected in compiler optimizations, there is a strong need for a practical solution to the problem of rapidly optimizing a program on a complex processor architecture. For that purpose, we outline our approach based on formalizing and systematizing the experience gained on program optimizations into a decision tree to drive end-user optimization tasks.

REFERENCES

[1] Susan L. Graham, David F. Bacon, and Oliver J. Sharp. *Compiler transformations for high-performance computing*. ACM Computing Survey, 1994.

[2] Edward E. Rothberg, Monica S. Lam, and Michael E. Wolf. *The cache performance and optimizations of blocked algorithms*. ASPLOS-IV, Santa Clara, Californina*, 1991*

[3] Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon , Alvin R. Lebeck. *Exact analysis of the cache behavior of nested loops*. PLDI*,* Snowbird, UTHA, 2001.

[4] Steve M. Carr. *Combining optimization for cache and instruction-level parallelism*. PACT, Boston, USA, 1996.

[5] Nicholas Mitchell, Larry Carter, Jeanne Ferrante, Karin Hogstedt. *Quantifying the multi-level nature of tiling interactions*. 10th Workshop on Languages Compilers for Parallel Computing, Minnesota, 1997.

[6] A.J. Smith. *Cache memories*. *ACM Computer Survey*, 1982

[7] Sharad Malik, Somnath Ghosh, Margaret Martonosi. *Precise miss analysis for program transformations with caches of arbitrary associativity.* ASPLOS-VIII, San Jose California, 1998.

[8] Robert H. Kuhn Bruce Leasure and Sanjiv M. Shah. *The KAP Parallelizer for DEC Fortran and DEC C Programs.* Digital Technical Journal vol.6 N°3, Summer 1994.

[9] Kathryn S. McKinley and Olivier Temam. *A quantitative analysis of loop nest locality*. ASPLOS-VII, Cambridge, Massachusetts, 1996

[10] Kathryn S. McKinley and Stephanie Coleman. *Tile size selection using cache organization and data layout.* PLDI*,* La Jolla, California, 1995.

[11] T. Kisuki, P.M.W. Knijnenburg, M.F.O. O'Boyle, H.A.G Wijshoff. *Iterative compilation in program optimization.* CPC, Aussois, France, 2000.

[12] James E. Smith and Gurindar S. Sohi. *The microarchitecture of superscalar processors*. Proceedings of IEEE, 1995

[13] Harsh Sharangpani. *Itanium MicroArchitecture Design.* Microprocessor Forum 1999.

[14] O. Temam, E. Granston and W. Jalby. *To copy or not copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts.* Supercomputing, Portland, USA, 1993.

[15] Juan J. Navarro, Toni Juan, and Tomás Lang. *MOB forms: A class of block algorithms for dense linear algebra operations*. Supercomputing, Washington, USA, 1994

[16] Larry Carter, Jeanne Ferrante, Susan Flynn Hummel, Bowen Alpern. *Hierarchical Tiling for improved superscalar performance. 9th* International Parallel Processing Symposium, Santa Barbara, California, 1995. (See also : *Hierarchical tiling : A methodology for high performance*. Technical report UCSD, 1996).

[17] T. Austin, D. Ernst, E. Larson, C. Weaver. *Simple Scalar Tutorial.* MICRO-30, North Carolina, USA 1997.

[18] M.E. Wolf, M.S. Lam. *A data locality optimizating algotrithm* , PLDI, Toronto, Ontario, Canada, 1991.

[19] T. Kisuki, P.M.W. Knijnenburg, M.F.O. O'Boyle. *Combined selection of tile sizes and unroll factors using iterative compilation*. PACT, Philadelphia, Pensylvania, 2000.

[20] G. Lindenmaier, K. S. McKinley, and O. Temam. *Load Scheduling using hardware counters*, Euro-Par, Munich, Gremany, 2000

[21] G. Fursin, M. O'Boyle, O. Temam, Gregory Watts. *Fast and Accurate Evaluation of Memory Performance Upper-Bound*. Proceedings of CPC, Edinburgh, Scotland, UK, 2001.

[22] C.W. Tseng, G. Rivera. *A comparison of compiler tiling algorithms*. CC, Amsterdam, Netherlands, 1999.

[23] S. Carr, Y. Guan. *Unroll-and-Jam using uniformly generated sets*. MICRO-30, North Carolina, USA, 1997.

[24] K.S. Gathin, L. Carter. *Memory Hierarchy Considerations for Fast Transpose and Bit-Reversals*. HPCA, Orlando, FL, USA, 1999.

[25] J. Bilmes, K Asanovic, C-W Chin, Jim Demmel. *Optimizing Matrix Multiply using PHiPAC : a Portable High-Performance*, *ANSI C Coding Methodology*. Supercomputing, San Jose, California, 1997.

[26] Kazushige Goto. *Optimized Libraries for ALPHA.* http://members.jcom.home.ne.jp/kgoto/

[27] *Compaq extended Math Library Reference Guide.*

[28] Srikanth T. Srinivasan and Alvin R. Lebeck, *Load Latency Tolerance In Dynamically Scheduled Processors*, ACM/IEEE International Symposium on Microarchitecture (MICRO), November 1998, Dallas, Texas.

[29] A.-T. Nguyen and al. *Accuracy and Speed-Up of Parallel Trace-Driven Architectural Simulation* 11th Int'l Parallel Processing Symposium, Geneva, Switzerland, 1997.

[30] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.-W. Liao, E. Bugnion, and M.S. Lam. *Maximizing Multiprocessor Performance with the SUIF Compiler.* IEEE Computer 1996.

[31] T. Kisuki P.M.W. Knijnenburg M.F.O. O'Boyle. *The Effect of Cache Models on Iterative compilation for Combined Tiling and Unrolling*. 3th Workshop on Feedback-Directed and Dynamic Optimization, Monterey, California, 2000

[32] T. Mowry and A. Gupta, "*Tolerating latency through software-controlled prefetching in shared-memory multiprocessors*", Journal of Parallel and Distributed Computing, June 1991.

[33] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. *Design and Evaluation of a Compiler Algorithm for Prefetching*, Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, October 1992, Boston, Massachusetts.