Extending Modulo Scheduling with Memory Reference Merging

Benoît Dupont de Dinechin

ST Microelectronics, CMG/MDT Division
Benoit.Dupont-de-Dinechin@st.com

Abstract. We describe an extension of modulo scheduling, called “memory reference merging”, which improves the management of cache bandwidth on microprocessors such as the DEC Alpha 21164. The principle is to schedule together memory references that are likely to be merged in a read buffer (LOADs), or a write buffer (STOREs). This technique has been used over several years on the Cray T3E block scheduler, and was later generalized to the Cray T3E software pipeliner. Experiments on the Cray T3E demonstrate the benefits of memory reference merging.

1 Introduction

As a result of the increasing gap between microprocessor processing speed, and memory bandwidth, cache optimization techniques play a major role in the performance of scientific codes. Cache optimization, like many other performance-improving optimizations, can be classified as high-level, or as low-level:

- High-level cache optimizations apply on a processor-independent program representation. High-level cache optimizations include loop restructuring transformations such as distribution, fusion, blocking, tiling, unimodular transformations [19], and unroll-and-jam [2].
- Low-level cache optimizations occur after instruction selection, when the program is represented as symbolic assembly code with pseudo-registers. The main motivation for low-level cache optimizations is that they may cooperate closely with instruction scheduling.

An important cache optimization, which is applied at high-level, low-level, or both, is prefetching / preloading [20] (non-binding / binding prefetching [10]):

- Prefetching describes the insertion of an instruction that has no architectural effects, beyond providing a hint to the hardware that some designated data should be promoted to upper levels of the memory hierarchy.
- Preloading consists in the execution of some LOAD instructions early enough so that the corresponding datum has time to move through the memory hierarchy up to a register before its value is actually used.

A typical application of preloading as a low-level loop optimization is the static prediction of hit / miss behavior, so as to provide instruction schedulers with realistic LOAD latencies [4].

In the case of the Cray T3E computer, the interplay between instruction scheduling (block scheduling and software pipelining), and low-level cache optimizations, appears as a challenging problem. The processing nodes of this machine are based on DEC Alpha 21164 microprocessors, backed up by a significant amount of Cray custom logic, including hardware stream prefetchers [25]. As we developed the software pipeliner of the Cray T3E [5,6,7], we assumed that the subject of low-level cache optimizations would reduce to: insertion of prefetching instructions, conversion of some LOADs into preloading instructions, and scheduling of these with the suitable latency.

However, a number of prefetching and preloading experiments conducted on the Cray T3E revealed a complex behavior: neither the insertion of prefetching instructions, nor preloading with the off-chip access latency, yield consistent results [14]: a few loops display significant performance improvements, while most others suffer performance degradations. Currently, software prefetching on the Cray T3E is limited to library routines that were developed and optimized in assembly code. Compiler preloading amounts to scheduling LOADs that are assumed to miss with the level-2 (on-chip) cache latency.

Among the explanations is the fact that prefetching and preloading consume additional entries in a read buffer called “Miss Address File” on the DEC Alpha 21164. For a number of codes, MAF entries end up being the critical resource of inner loops. Performance of such loops used to be quite unpredictable, as merging of memory references in the read buffer and the write buffer happened as an uncontrolled side-effect of instruction scheduling. On these loops, block scheduling¹ would often perform better than software pipelining, because the latter technique spreads unrolled memory references across the loop schedule, eventually resulting in a lower amount of memory reference merging.

In this paper, we describe the “memory reference merging” optimization, which evolves from the simple “LOAD grouping” technique originally implemented in the block scheduler of the Cray T3D by Andrew Meltzer. Section 1 states the performance problem presented by memory hierarchies such as found on the DEC Alpha 21164, which motivates the memory reference merging optimization. Section 2 describes the design and implementation of this technique in the scheduling engine of the Cray T3E software pipeliner. Section 3 presents some of the experimental results of the memory reference merging optimization, obtained with the current Cray T3E production compilers.

2 Motivating Problem

2.1 Read and Write Buffers of the DEC Alpha 21164

Modern microprocessors have non-blocking data caches [9]: a LOAD instruction that misses in cache does not prevent subsequent instructions from be-

¹ On the Cray T3E this includes “bottom-loading”, a loop pipelining technique where the loop body is rotated before block scheduling so as to schedule LOADs with a longer latency.

ing issued, at least up to the first read reference to the LOAD's destination register. A restricted support for non-blocking LOADs was found on the DEC Alpha 21064 microprocessor, whose "load silo" could hold two outstanding level-1 cache misses, while servicing a third LOAD hit. In the DEC Alpha 21164 microprocessor [8], the load silo has been generalized to a Miss Address File, which is in fact a MSHR (Miss Status Holding Register) file as defined by Kroft [15].

More precisely, the DEC Alpha 21164 microprocessor includes [11]: a memory address translation unit or "Mbox"; a 8 KB level-1 data cache (Dcache), which is write-through, read-allocate, direct-mapped, with 32-byte blocks; a 96 KB level-2 instruction and data cache (Scache), which is write-back, write-allocate, 3-way set-associative, and is configured with either 32-byte or 64-byte blocks. The Dcache sustains two reads or one write per processor cycle. Maximum throughput of the Scache is one 32-byte block read or write per two cycles.

The Mbox itself contains several sections: the Data Translation Buffer (DTB), the Miss Address File (MAF), the Write Buffer (WB), and Dcache control logic [11]. When a LOAD instruction executes, the virtual address is translated, and Dcache access proceeds. In case of a Dcache miss, the six entries of the MAF are associatively searched for an address match at the 32-byte block granularity. If a match is found, and other implementation-related rules are satisfied, the new miss is merged in the matching MAF entry. If no match is found, or if merging cannot take place due to some reason, a new entry is allocated in the MAF. In case the MAF is full, the processor stalls until a new entry is available.

On the DEC Alpha 21164, the MAF is the read counterpart of a write buffer [26]: its purpose is to buffer unsatisfied read requests directed to a given 32-byte cache block, so that they can all be served in a single 2-cycle transaction when data returns from the Scache (or from external memory). Indeed, the WB on the DEC Alpha 21164 also maintains a file of six associative 32-byte blocks, and may store any of them to the Scache in a single 2-cycle transaction. Write buffer merging rules are less constrained than MAF merging rules though, mainly because a WB entry does not have to maintain a list of destination registers.

The performance implications of the MAF are quite significant: assuming all data read by a loop fits into the Scache, and that the access patterns and the instruction schedule are such that four 64-bit LOAD misses merge in every MAF entry, then the loop runs without memory stalls, provided that LOADs are scheduled with a suitable latency. On the other hand, if MAF merging is not exploited due to poor spatial locality, or because the LOADs to the same cache blocks are scheduled too many cycles apart, or due to some MAF implementation restriction, then the memory throughput cannot be better than one LOAD every other cycle, a fourfold decrease compared to Dcache bandwidth.

The ability to run at full speed when data is in Scache and not in Dcache is especially important on the DEC Alpha 21164. Indeed, there is no guarantee that floating-point data ever reaches Dcache when it is LOAded into a register from the Scache. The reason is that floating-point Dcache refills have the lowest priority when it comes to writing into Dcache [11]. In fact, for instruction

scheduling purposes, Dcache behavior regarding to floating-point data is so unpredictable that it is better to assume it is not there [14]. In this respect the DEC Alpha 21164 appears quite similar to the MIPS R8000, whose memory system bypasses level-1 cache in case of floating-point references [12].

Just like the MAF, correct exploitation of the write buffer on DEC Alpha like processors yields significant performance improvements, as reported in [26].

2.2 Problem Statement

As discussed in the previous section, careful exploitation of the merging in a read buffer and/or a write buffer, such as the MAF and the WB of the DEC Alpha 21164, can significantly improve performance. The stage of compilation where such exploitation best takes place is instruction scheduling, as merging behavior ultimately depends on the relative issue dates of the mergeable instructions.

In this paper, we shall focus on modulo scheduling [21,16,3,22], a widely used software pipelining technique which encompasses block scheduling as a degenerate case. Modulo scheduling is a software pipelining technique where all the loop iterations execute the same instruction schedule, called the *local schedule* [22,16], and such that the execution of any two successive iterations is separated by a constant number of cycles, called the *Initiation Interval (II)*. The general process of modulo scheduling can be summarized as follows:

1. Compute the lower bound *recMII* on the *II*, which makes scheduling possible as far as recurrence constraints are concerned. Compute the lower bound *resMII* on the *II* set by the resource constraints. This provides an initial value $\min(\text{recMII}, \text{resMII})$ for the *II*.
2. Schedule the instructions, subjected to: (a) the modulo resource constraints at the current *II* must not be violated; (b) each instruction must be scheduled within its *margins*, that is, its current earliest and latest possible schedule dates (called *Estart* and *Lstart* by Huff [13]).

When all the instructions can be scheduled this way, the local schedule of the software pipeline is obtained. From this local schedule, the complete software pipeline code is constructed [23,7].

In case the current instruction to schedule modulo resource conflicts with the already scheduled instructions for all dates within its margins, a *failure condition* is detected. Although failure at the current *II* can be handled in a variety of ways, the last resort is to increase the *II*, and to restart scheduling from scratch. Eventually this strategy succeeds, as modulo scheduling degenerates to block scheduling when the *II* grows large enough.

Within the framework of modulo scheduling and block scheduling, we state the problem of memory reference merging as follows:

- Compute the *merge intervals* associated to pairs of memory references. A merge interval is defined to contain the relative issue dates of the two references in the pair such that merging is possible, assuming that an entry is available in the read buffer (LOADs) or the write buffer (STOREs).

- While scheduling instructions, assume suitable read buffer or write buffer resource conflicts for any pair of mergeable memory references whose relative issue dates do not belong to one of the merge intervals of that pair.

Still the purpose of the instruction scheduler is to reduce execution times, either by minimizing the length of the instruction schedule (block scheduling), or by maximizing the loop throughput (software pipelining).

Although motivated by the DEC Alpha 21164 memory hierarchy, memory reference merging has wider applications than optimizing instruction schedules for that microprocessor. In particular, the recently available DEC Alpha 21264 microprocessor extends the number of entries in the read buffer and the write buffer to eight, while the block size of the individual buffer entries increases from 32 to 64 bytes. Unlike the Scache of the DEC Alpha 21164, the level-2 cache of the DEC Alpha 21264 is not on-chip. As a result, memory reference merging is likely to become a key optimization on the DEC Alpha 21264.

On the MIPS R8000, best performing instruction schedules are obtained by “pairing” memory references. The principle of pairing is to allow dual-issuing of memory references only in cases they are guaranteed to access distinct cache banks [27]. Pairing is implemented in the MIPSpro heuristic-based modulo scheduler, which betters the MIPS R8000 optimum integer linear programming modulo scheduler [24] by as much as 8% geometric performance mean on SPEC FP benchmark. It was not until after the latter was extended to include pairing that performance of the two software pipeliners became comparable [27]. As we shall see, pairing is a special case of memory reference merging.

Other simple forms of memory reference merging apply to other processors, such as fusing pairs of LOADs on the IBM POWER 2 to take advantage of the higher cache bandwidth available from quad-word (128 bits) LOAD instructions.

3 Memory Reference Merging

3.1 Approximating Merge Intervals

Memory reference merging is an optimization that applies during instruction scheduling of inner loops. At this stage of the compilation process, memory references can be partitioned into three classes:

- memory references whose effective address is a loop-invariant expression;
- memory references whose effective address is an inductive expression with a compile-time constant step (*simple inductions* [7]);
- memory references whose effective address is a complex loop-variant expression, including inductive expressions with a symbolic step.

Let k be the normalized loop counter of the inner loop. We shall abstract the effective address of memory reference i in the loop body as $(r_i + k\delta_i + o_i)$, where r_i is a base value, δ_i is the induction step, and o_i is a constant offset. In cases of complex loop-variant effective address expressions, we shall assume

that the induction step value δ_i is $\perp$ (undefined). Merge buffer entry block size is denoted b (32 bytes on the DEC Alpha 21164). All the quantities r_i, δ_i, o_i, b are expressed in the lower addressable unit of the machine, typically a byte.

The first step of memory reference merging is to partition the memory references of the loop body into *base groups*. A base group contains all the references that can share the same base value and induction step, perhaps after some adjustments of the offset values. As a matter of illustration, let us consider the following cases, where the value of j is unknown at compile-time:

<pre>while (i < n) { ... = a[i]; i = i + 1; ... = a[i-3]; }</pre>	<pre>while (i < n) { ... = a[i]; i = i + j; ... = a[i-3]; }</pre>	<pre>while (i < n) { ... = a[i]; ... = a[i-3]; i = i + j; }</pre>
--	--	--

In the first case, the two memory references to `a` belong to the same base group, with respective offsets 0 and -2. In the second case, the unknown value of j forces the two memory references into different base groups. In the third case, even though the induction step is a compile-time unknown, the two memory references to `a` can again be folded into the same group, with offsets 0 and -3.

Once base groups are available, we define the *mergeable* memory references as those that will use an entry in the read buffer or the write buffer. On the DEC Alpha 21164, this rules out Dcache hits, which we assume only for integer LOADs with a loop-invariant effective address. Then, among the mergeable memory references, we identify the *incompatible* pairs:

- Any two memory references from different base groups do not merge.
- A LOAD and a STORE cannot merge, since they go respectively to the read buffer (MAF on the DEC Alpha 21164), and to the write buffer.
- The read buffer may rule out merging for implementation-related limitations. For instance the MAF of the DEC Alpha 21164 prevents LOADs of different data sizes, data types, or 4-byte alignment, from merging [11].

The merge intervals of incompatible pairs are obviously the empty set. For the other (compatible) pairs, we assume that merging is possible provided that:

- (1) the two effective addresses fall in the same merge buffer entry, which is an aligned block of b addressable units;
- (2) in the case of LOADs, the two effective addresses must not refer to the same word², which is an aligned block of w addressable units;
- (3) the two issue dates t_i and t_j are no more than m cycles apart.

This value m represents the upper limit on the number of cycles below which the hardware may consider two memory references as candidates for merging.

² This is an implementation-related restriction. On the DEC Alpha 21164, w is 8 bytes.

Absolute alignments are impractical to manage at compile-time, since the number of cases to consider grows exponentially with the number of memory reference streams in the loop. Thus we approximate conditions (1) and (2) as:

- (1') the two effective addresses differ by no more than b addressable units.
- (2') the two LOAD effective addresses differ by more than w addressable units.

Although conditions (1'), (2'), and (3), are accurate enough for the purposes of block scheduling, they miss many opportunities of merging in the case of modulo scheduling. Indeed modulo scheduling works by overlapping the execution of successive loop iterations. For any two memory references i and j of the loop body, we must account for the possibility of merging between $i_{k'}$ and $j_{k''}$, where $i_{k'}$ and $j_{k''}$ denote respectively the k' -th and k'' -th instances of memory references i and j . Conditions (1'), (2'), (3) then become:

- (a) $(r + k''\delta + o_j) - (r + k'\delta + o_i) \in [-b + 1, b - 1]$
($r_i = r_j = r$, and $\delta_i = \delta_j = \delta$, since i and j are in the same base group)
- (b) $(r + k''\delta + o_j) - (r + k'\delta + o_i) \notin [-w + 1, w - 1]$
(assume $w = 0$ in case of STOREs, so that $[-w + 1, w - 1] = \emptyset$)
- (c) $(t_j + k''\lambda) - (t_i + k'\lambda) \in [-m + 1, m - 1]$
(λ is the current value of the initiation interval of the software pipeline)

Computing merge intervals for memory references i and j now reduces to:

- Find the set K_{ij} of all values $k \stackrel{\text{def}}{=} k' - k''$ that satisfy (a), (b), and (c).
- The merge intervals are $[k\lambda - m + 1, k\lambda + m - 1] \cap [t_{ij}^-, t_{ij}^+]$, $k \in K_{ij}$.

Here $[t_{ij}^-, t_{ij}^+]$ denotes the admissible range of $t_j - t_i$ while scheduling instructions.

The set K_{ij} associated to a memory reference pair (i, j) is actually straightforward to compute using interval arithmetic. Let us first define, for any integers a, b, c , the *reduce* operation on interval $I \stackrel{\text{def}}{=} [a, b]$ as:

$$\text{reduce}([a, b], c) \stackrel{\text{def}}{=} \begin{cases} c > 0 & \rightarrow \left[\lceil \frac{a}{c} \rceil, \lfloor \frac{b}{c} \rfloor \right] \\ c < 0 & \rightarrow \left[\lceil \frac{-b}{-c} \rceil, \lfloor \frac{-a}{-c} \rfloor \right] \\ c = 0 \wedge a \leq 0 \wedge b \geq 0 & \rightarrow] -\infty, +\infty[\\ c = \perp \wedge a \leq 0 \wedge b \geq 0 & \rightarrow [0, 0] \\ \text{default} & \rightarrow \emptyset \end{cases}$$

In other words, $I' \stackrel{\text{def}}{=} \text{reduce}(I, c)$ is the maximum interval such that $cI' \subseteq I$. The reduce operation allows to compute K_{ij} in just four simple steps:

- $I_o^- \stackrel{\text{def}}{=} [o_j - o_i - b + 1, o_j - o_i - w]$
(I_o^- contains all the $k\delta$ such that $o_j - o_i - k\delta \in]w - 1, b - 1]$)
- $I_o^+ \stackrel{\text{def}}{=} [o_j - o_i + w, o_j - o_i + b - 1]$
(I_o^+ contains all the $k\delta$ such that $o_j - o_i - k\delta \in [-b + 1, -w + 1[$)

- $I_d \stackrel{\text{def}}{=} [t_{ij}^- - m + 1, t_{ij}^+ + m - 1]$
 (I_d contains all the $k\lambda$ such that $t_j - t_i - k\lambda \in [-m + 1, m - 1]$)
- $K_{ij} \stackrel{\text{def}}{=} (\text{reduce}(I_o^-, \delta) \cup \text{reduce}(I_o^+, \delta)) \cap \text{reduce}(I_d, \lambda)$

It is easy to check that K_{ij} computed this way is tight, that is, it does not contain values of $k \stackrel{\text{def}}{=} k' - k''$ which do not satisfy conditions (a), (b), and (c).

An important property is that we are able to derive non-empty K_{ij} even if the induction step δ is undefined ($\perp$). In that case, only memory reference pairs from the same iteration may merge together, a condition which is good enough for many of the loops unrolled by the high-level optimizer. The main point of handling undefined induction steps however is that the technique now applies to block scheduling as well. All that is required in this case is to carry the K_{ij} computations with a suitably large value of λ .

3.2 Modulo Scheduling Extensions

According to the description of modulo scheduling given in section 2.2, the integration of memory reference merging involves the following extensions:

- Introduce a single “Cache Bandwidth” (CB) resource in the processor modelization. Extend the reservation tables so that all mergeable memory references use the CB resource for some time. Only two mergeable memory references that are scheduled within one of the associated merge intervals may use the CB resource at the same time. This CB resource modelizes the bandwidth limit between the two levels of the memory hierarchy where read buffers and write buffers operate.

First, we considered introducing resources to represent individual read buffer and write buffer entries. However, the primary purpose of these buffers is to supply a continuous flow of read or write requests to the lower level of the memory hierarchy (the Scache on the DEC Alpha 21164). As long as these buffers do not overflow, the single CB resource accurately represents the run-time behavior of the memory system. In our implementation, a mergeable memory reference reserves CB for 2 cycles.

- When computing $resMII$, take into account the fact that some memory references will be merged. Compared to modulo scheduling without memory reference merging, the only difference is the need to introduce the minimum possible use of the CB resource as a lower bound for $resMII$.

The minimum possible use of the CB resource is computed by constructing the *merge sets*, defined as all the subsets of the equivalence classes of the “merge intervals are not empty” binary relation. These sets are weighted by their minimum collective use of the CB resource, and the total minimum use of the CB resource is obtained as the solution of the so-called “Weighted Vertex Cover” problem [1]

An exact computation of the minimum possible use of the CB resource could be expensive, but is not required. Indeed this value only impacts the computation of $resMII$. Assuming zero as the minimum possible use of the CB

resource when computing *resMII* only enables the modulo scheduler to attempt modulo scheduling at values of the *II* that are too low to succeed.

- In case of a heuristic modulo scheduler, when selecting a mergeable memory reference as the current instruction to schedule, we first try the schedule dates that intersect the maximum number of merge intervals from the pairs whose first member is the current instruction, and whose second member is an already scheduled memory reference instruction.

Indeed heuristic modulo schedulers make decisions at two levels: what is the next instruction to schedule, and what are the schedule dates to try first. In our implementation, memory reference merging only impacts the first decision in an indirect way: our scheduling priority function weights the not yet scheduled instructions by adding their use of the loop critical resource, and by subtracting their scheduling slack [13]. Trying the schedule dates where merging will occur first is a very light modification of the scheduling engine that produces very satisfactory results.

4 Experimental Results

The experimental results presented in this section are performance numbers obtained in dedicated mode³ on a Cray T3E-600. On this first-generation Cray T3E, the microprocessor clock frequency is 300 MHz, leading to a theoretical peak performance of 600 MFLOPS per processing node. The more recent Cray T3E-900 presents several improvements over the Cray T3E-600, including versions of the DEC Alpha 21164 microprocessor with the redesigned DEC EV-56 logic core, and a microprocessor clock frequency increased to 450 MHz.

In order to better appreciate our performance numbers, let us first provide some quantitative data about the memory bandwidth problem on machines powered by a DEC Alpha 21164 such as the Cray T3E. Typical scientific applications are known to require a balance of about two LOADs and one STORE per floating-point multiplication-addition. By tabulating the floating-point operations (FLO) per cycle scaled from the bandwidth by 2/3, we obtain a realistic upper bound on the floating-point performance of a typical scientific application:

Level	Words / Cycle	Scaled FLO / Cycle	Restrictions
Dcache	2	1.33	Direct-Mapped
Scache	2	1.33	32-byte Block
Streams	.53	.35	6 Streams
E-registers	.33	.22	Distribution
DRAM Page Hit	.31	.20	1-Entry Buffer
DRAM Page Miss	.21	.14	Local Memory

For more details about the various levels of the Cray T3E memory hierarchy, please refer to [25]. Although the DEC Alpha 21164 of a Cray T3E-600 is

³ Special thanks to Sean Palmer, Tuyet-Anh Tran, and Anand Singh, from Cray Research, for preparing and running these experiments.

rated at 600 MFLOPS peak, a 2/3 ratio between floating-point operations and memory accesses imply that the performance upper bound is more in the range of 450 MFLOPS under the best cases of Scache bandwidth exploitation. For applications that need to access memory off-chip, an upper bound of 84 MFLOPS is expected in cases the accesses hit the DRAM in non-paged mode.

When running the experiments, the Cray f90 Fortran compiler was used with options `-Ounroll12 -Opipeline2`. The `-Ounroll12` option enables automatic unrolling by a compiler-selected amount, typically 4. Unrolling by 4 is a good overall tradeoff, as it creates groups of 4 memory references which could theoretically merge in a single MAF or WB entry, in the case of dense 64-bit access streams. Higher levels of unrolling can be specified at the source code level by inserting compiler directives. Unrolling by 8 potentially yields higher performance, but significantly increases the chance that the software pipeliner runs out of registers. When this happens, the loop is not software pipelined.

Fig. 1. Livermore loops.

The first set of results, displayed in figure 1, was obtained by running the `livkern.f` program (9/OCT/91 version mf523), also known as the Livermore loops. Although performance numbers are collected by this program for 24 kernels, we only include results for the loops that were successfully software pipelined. In particular kernels 7, 8, 9, 13, are not software pipelined under `-Ounroll12`, due to high register pressure. The other reason some of the kernels are not software pipelined is the lack of IF-conversion in the Cray T3E compiler. In figure 1, memory reference merging is effective on loops K1, K10, and K12.

Fig. 2. Linpack benchmark.

The second set of results, displayed in figure 2, comes from the “Linpack” benchmark by Dongarra. Among the 19 loops in this program that do not contain subroutines calls nor conditionals, 17 were software pipelined. A geometric mean performance improvement of 1.41 was obtained, and is explained by the fact that all data referenced is in Scache, while all the pipelined loops are parallel or vector with dense access streams. Under these conditions, memory reference merging is able to exploit most of the Scache bandwidth available.

The third set of results appears in figure 3, which displays the percentages of relative performance improvement of memory reference merging over the non-merging case for different applications from the Applied Parallel Research suite. Here again some improvements are significant, in particular for programs **X42** (351 lines, 21.75% increase), and **APPSP** (4634 lines, 9.5% increase). In some cases such as **SCALGAM** and **SHALLOW77**, performance is degraded by as much as 2%. We could trace these degradations to MAF entry full conditions, which are triggered because memory reference merging only manages *bandwidth* between Dcache and Scache, and not the number of read or write buffer entries.

5 Related Work

The principle of memory reference merging is to compute and associate “merge intervals” to “mergeable” memory reference pairs, and to introduce a single “Cache Bandwidth” (CB) resource in the processor modelization. The modulo scheduler is then extended to assume pairwise conflicts on the CB resource for all mergeable memory references, except for those that are scheduled within one of the merge intervals associated to the mergeable memory reference pair.

Fig. 3. Applied Parallel Research suite % relative improvements.

Computing the merge intervals bears similarities with *reuse analysis*, that is, identifying the number of distinct cache blocks referenced in a loop, in order to guide locality-enhancing loop restructuring [19], or to insert a minimum number of prefetch instructions [20]. Although we only perform reuse analysis at the innermost loop level, we are able to handle compile-time unknown induction steps, whereas traditional techniques require compile-constant induction steps. Work reported in [4] applies reuse analysis to the static prediction of hit / miss behavior. It can be seen as a preliminary step for memory reference merging, as we do not consider LOAD hits as mergeable (they do not use the CB resource).

Following an innovative line of research, López, Valero, Llosa, and Ayguadé [18,17] develop a compilation technique where loop bodies are unrolled, and the memory accesses are packed in case the target architecture supports “widened” data and memory paths. The resulting scheduling graph is then scheduled using a traditional modulo scheduler. The primary difference between “compaction of memory accesses” [18] and memory reference merging, is that the technique of López et al. operates before modulo scheduling, and requires a specific architectural support: packed versions of arithmetic and memory instructions.

More related to our work is the “memory bank pairing” optimization formulated by Stoutchinin [27]. This technique avoids level-2 cache bank conflicts on the MIPS R8000 by preventing two loads that may access the same cache bank from issuing at the same cycle. This optimization appears as a form of memory reference merging with single-cycle merge intervals ($m = 1$), once the constant offsets of the effective addresses are reduced modulo 16. This work is significant

to us, as it demonstrates that a simpler variant of the memory reference merging problem can be formulated and solved in the framework of optimal modulo scheduling using integer linear programming.

6 Conclusion

This paper describes the design, implementation, and experimental results, of the memory reference merging extension of the Cray T3E modulo scheduler. This technique takes advantage of the read and write buffers present at the higher levels of a memory hierarchy, specifically the MAF and the WB of the DEC Alpha 21164 microprocessor. Experiments demonstrate that memory reference merging is quite effective: on the Cray T3E-600 production compiler, it improves the geometric mean performance of the Linpack benchmark by as much as 41%.

More generally, memory reference merging improves block scheduling and modulo scheduling on processors whose memory hierarchy favors a regular form of coupling between spatial (effective addresses) and temporal (schedule dates) locality of memory references: read and write buffers (DEC Alpha 21164, DEC Alpha 21264), and multi-banked interleaved caches (MIPS R8000). Memory reference merging also provides the proper foundations for automatic packing of memory references (IBM POWER 2, “widened” [18] processors).

References

1. R. BAR-YEHUDA, S. EVEN: *A Linear-Time Approximation for the Weighted Set Cover Problem* Journal of Algorithms, Vol. 2, 1981. 281
2. S. CARR, Y. GUAN: *Unroll-and-Jam Using Uniformly Generated Sets* Micro-30 – Proceedings of the 30th International Symposium on Microarchitecture, Dec. 1997. 274
3. J. C. DEHNERT, R. A. TOWLE: *Compiling for Cydra 5* Journal of Supercomputing, vol. 7, pp. 181–227, May 1993. 277
4. C. DING, S. CARR, P. SWEANY: *Modulo Scheduling with Cache-Reuse Information* Proceedings of EuroPar’97, LNCS #1300, Aug. 1997. 274, 285
5. B. DUPONT DE DINECHIN: *Insertion Scheduling: An Alternative to List Scheduling for Modulo Schedulers* LCPC’96 – 8th International Workshop on Languages and Compilers for Parallel Computing, LNCS #1033, Columbus, Ohio, Aug. 1995. 275
6. B. DUPONT DE DINECHIN: *Parametric Computation of Margins and of Minimum Cumulative Register Lifetime Dates* LCPC’97 – 9th International Workshop on Languages and Compilers for Parallel Computing, LNCS #1239, San Jose, California, Aug. 1996. 275
7. B. DUPONT DE DINECHIN: *A Unified Software Pipeline Construction Scheme for Modulo Scheduled Loops* PaCT’97 – 4th International Conference on Parallel Computing Technologies, LNCS #1277, Yaroslavl, Russia, Sep. 1997. 275, 277, 278
8. *Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor* Digital Technical Journal, Vol. 7, No. 1, Jan. 1995. 276
9. KEITH I. FARKAS, NORMAN P. JOUPPI: *Complexity/Performance Tradeoffs with Non-Blocking Loads* WRL Research Report 94/3, Western Research Laboratory, Mar. 1994. 275

10. A. GUPTA, J. HENNESSY, K. GHARACHORLOO, T. MOWRY, W.-D. WEBER: *Comparative Evaluation of Latency Reducing and Tolerating Techniques* ISCA'91 – 18th International Symposium on Computer Architecture, May 1991. 274
11. *Alpha 21164 Microprocessor Hardware Reference Manual*, Document EC-QAEQB-TE, Digital Equipment Corporation. 276, 279
12. P. Y.-T. HSU: *Design of the R8000 Microprocessor* IEEE Micro, 1993. 277
13. R. A. HUFF: *Lifetime-Sensitive Modulo Scheduling* PLDI'93 – Conference on Programming Language Design and Implementation, June 1993. 277, 282
14. R. E. KESSLER: *Livermore Loops Single-Node Code Optimization for the CRAY T3E* Technical Report, System Performance Group, Cray Research Inc., 1995. 275, 277
15. D. KROFT: *Lockup-Free Fetch/Prefetch Cache Organization* ISCA'81 – 8th International Symposium on Computer Architecture, May 1981. 276
16. M. LAM: *Software Pipelining: An Effective Scheduling Technique for VLIW Machines* PLDI'88 – Conference on Programming Language Design and Implementation, 1988. 277
17. D. LÓPEZ, J. LLOSA, M. VALERO, E. AYGUADÉ: *Resource Widening Versus Replication: Limits and Performance-Cost Trade-off* ICS-12 – 12th International Conference on Supercomputing, Melbourne, Australia, July 1998. 285
18. D. LÓPEZ, M. VALERO, J. LLOSA, E. AYGUADÉ: *Increasing Memory Bandwidth with Wide Buses: Compiler, Hardware and Performance Trade-offs* ICS-11 – 11th International Conference on Supercomputing, Vienna, Austria, July 1997. 285, 286
19. K. MCKINLEY, S. CARR, C.-W. TSENG: *Improving Data Locality with Loop Transformations* ACM Transactions on Programming Languages and Systems, Vol. 18, No. 4, Jul. 1996. 274, 285
20. T. C. MOWRY, M. S. LAM, A. GUPTA: *Design and Evaluation of a Compiler Algorithm for Prefetching* ASPLOS-V – Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA, 1992. 274, 285
21. B. R. RAU, C. D. GLAESER: *Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing* 14th Annual Workshop on Microprogramming, Oct. 1981. 277
22. B. R. RAU: *Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops* MICRO-27 – 27th Annual International Symposium on Microarchitecture, San Jose, California, Nov. 1994. 277
23. B. R. RAU, M. S. SCHLANSKER, P. P. TIRUMALAI: *Code Generation Schemas for Modulo Scheduled Loops* MICRO-25 – 25th Annual International Symposium on Microarchitecture, Portland, Dec. 1992. 277
24. J. C. RUTTENBERG, G. R. GAO, A. STOUTCHININ, W. LICHTENSTEIN: *Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler* PLDI'96 – Conference on Programming Language Design and Implementation, Philadelphia, PA, May 1996. 278
25. S. L. SCOTT: *Synchronization and Communication in the T3E Multiprocessor* ASPLOS-VII – Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, Oct. 1996. 275, 282

26. K. SKADRON, D. W. CLARK: *Design Issues and Tradeoffs for Write Buffers* HPCA'97 – Proceedings of the 3rd International Symposium on Computer Architecture, San Antonio, TX, Feb. 1997. [276](#), [277](#)
27. A. STOUTCHININ: *An Integer Linear Programming Model of Software Pipelining for the MIPS R8000 Processor* PaCT'97 – 4th International Conference on Parallel Computing Technologies, Yaroslavl, Russia, Sep. 1997. [278](#), [285](#)