Optimizing the Memory Bandwidth with Loop Fusion

Paul Marchal
IMEC/KULEUVEN
Heverlee, Belgium
marchal@imec.be

José Ignacio Gómez
DACYA U.C.M.
Madrid, Spain
jigomez@dacya.ucm.es

Francky Catthoor
IMEC/KULEUVEN
Heverlee, Belgium
catthoor@imec.be

ABSTRACT

The memory bandwidth largely determines the performance and energy cost of embedded systems. At the compiler level, several techniques improve the memory bandwidth at the scope of a basic block, but often fail to exploit all. We propose a technique to optimize the memory bandwidth across the boundaries of a basic block. Our technique incrementally fuses loops to better use the available bandwidth. The resulting performance depends on how the data is assigned to the memories of the memory layer. At the same time, the assignment also strongly influences the energy cost. Therefore, we combine in our approach the fusion and assignment decisions. Designers can use our output to trade-off the energy cost with the system's performance.

Categories and Subject Descriptors: B.3 Hardware-Memory strutures[Scratchpad memories] D.3.4 SoftwareProgramming languages[Optimisation]

General Terms: Algorithms, Performance, Design

Keywords: low power, loop fusion, memory bandwidth

1. INTRODUCTION

Two complementary techniques are often used to improve the memory bandwidth. On the one hand designers exploit a hierarchy of memory layers to hide the latency of the slow memories. On the other hand, they use multiple memories to increase the bandwidth on each layer.

In this paper, we focus on how to optimize the memory bandwidth from the memory layer closest to the processor. This layer contains multiple memories. Nowadays the compiler determines which memory operations are scheduled in parallel. In the simplest case, it puts as many memory operations in parallel as load/store units exist on the processor. It even schedules operations in parallel of which the corresponding data resides in the same single-port memory. Since those data cannot be accessed in parallel, the memory interface has to serialize these accesses, thereby stalling the processor. By distributing the data across multiple memories, we can execute more operations in parallel and avoid stalls [13]. The compiler can also optimize the instruction schedule to avoid simultaneous accesses to the same memory [17]. This can also be achieved with multiport memories, but this comes at an energy/area penalty.

How we distribute the data has an important impact on the energy cost. In low-power systems designers often combine different sized memories in a single memory layer (e.g., [1]). For a high performance, designers should distribute the data such that the compiler can schedule as many accesses as possible in parallel. As a result, many data structures are then usually assigned to energy inefficient memories. In contrast for a low energy cost, designers should cluster the frequently accessed data in the small, energy efficient memories. However, this restricts the number of accesses that the compiler can schedule in parallel and thus limits the system's performance. Thus, a trade-off exists between performance and energy consumption which current techniques can automatically explore [17].

Unfortunately, today's techniques only optimize the memory bandwidth per basic block. If we optimize the memory bandwidth across the boundaries of a basic block, we can obtain a higher performance for the same assignment, i.e. energy cost. A well known global transformation is loop fusion. By combining two loop bodies, fusion increases the number of independent memory operations in each basic block. Consequently, the compiler can better fill the memory access slots. The question, then, is which loop pairs we best fuse to optimize the memory bandwidth. Generally, fusing loops with many empty memory access slots delivers the largest performance gain. The data assignment determines which loops have most empty access slots and thus which loops should be fused. At the same time, data assignment also determines the energy cost (as explained above). The problem consists of finding the most energy efficient data assignment and the corresponding fusion decisions such that the applications' time-constraints are met.

To address the above problem, we present in this paper an integrated data assignment/fusion approach. We have constructed a source-to-source loop fusion tool and verified our approach on the Trimaran VLIW simulator.

This paper is organized as follows. First, we discuss the related work (Sect. 2), then we outline our approach with an example (Sect. 3). Subsequently, we explain our technique (Sect. 4-5) and finally, we quantify it (Sect. 6).

2. RELATED WORK

Memory optimization is a widely researched topic [10].

CODES+ISSS'04, September 8–10, 2004, Stockholm, Sweden.

Several techniques have been introduced to optimize the memory bandwidth. They mostly integrate memory bandwidth optimization within the compiler/linker [13][9][4]. In [13] the performance is maximized by distributing the data across the different memories such that as many accesses as possible can be executed in parallel. The authors of [9] and [4] propose an instruction scheduling technique. They expose the access latency of SDRAM memories to the compiler. The scheduling is done after the data has been assigned. [17] optimizes the memory bandwidth in a separate step before compilation. They optimize the schedule of the memory operations and the data assignment together. They output a (partial) data assignment which constrains the final instruction scheduling. It guarantees that enough memory bandwidth exists to meet the deadline, while remaining as energy-efficient as possible. Above techniques optimize the memory bandwidth within the boundaries of a basic block. This limits the performance if not enough independent memory operations are available in each basic block.

More global optimization techniques can further improve the performance. In the past, several authors have proposed techniques to globally schedule instructions to parallelize code [6]. They do not focus on how to optimize the memory bandwidth. [16] defines an operation schedule which reduces the number of memory ports. However, it does not take into account which data structures are accessed or how they are mapped in the memory. Furthermore, the resulting schedule is implemented with a dedicated controller, which is quite different from commonly used processors.

Loop transformations are an interesting alternative to optimize the code across the boundaries of a basic block. Originally, they have been developed to extract parallelism [8] increase regularity [3] and/or improve data locality [2]. One particular transformation, *loop fusion*, is often used to increase the size of a loop body. The compiler can then detect more instruction level parallelism [11]. This is also what we need for memory bandwidth optimization. However, current techniques can only fuse loops which execute the same number of iterations. In practice, this condition limits the applicability of fusion. Several authors combine loop fusion with loop shifting to increase its applicability (see [15] for an overview), but target a different optimization objective. They want to increase locality or the life-time of arrays. As a result, the mechanism to decide which loops to combine is different. We cannot thus directly reuse the above techniques in our context.

3. MOTIVATIONAL EXAMPLE

In this section we illustrate why our integrated data assignment/fusion outperforms existing memory bandwidth optimization techniques. The example consists of three data-dominated loops (see code in Fig. 1-left) which are executed on a platform that consists of three memory ports fully connected to three single-port memories: two 4kB ones (0.11nJ/access) and a 2kB one (0.06nJ/access).

Because the applications are data dominated, the duration of the memory access schedule determines the performance of the loops. We therefore assume that the remaining operations can be performed in parallel with the memory accesses or take only limited time. We now study the influence of loop fusion on the length of the memory access schedule and the energy cost.

Most compilers are unaware of the underlying memory

```
int A[301];int B[100];int D[100]
int C[100]; int U[2];
int i,j;

for (i=0; i<100; i++) // loop 1
 A[i+1] = A[i] + 1;

for (i=0; i<100; i++) // loop 2
 D[i] = C[i] + B[i];

for (i=0; i<2; i++){  // loop3
 for (j=0; j<40; j++)  // loop31
   D[j] = D[j-1]
            + D[j];
 U[i] = D[39];
}
```

```
int A[300];int B[100];int D[100]
int C[100]; int U[2];

for (int i=0; i<100; i++) // loop 2
 D[i] = C[i] + B[i];

for (int i=0; i<2; i++){ // loop 1&3
 for (int j=0; j<40; j++){
  D[j] = D[j-1]+ D[j];
  A[40*i+j] = A[40*i+j-1] + 1;
 }
 U[i] = D[39];
}
// remainder of loop 1
for (int i=0; i<20; i++)
 A[i+80] = A[i-1+80] + 1;
```

Figure 1: Motivational example: original code (left), code after fusion (right)

architecture. During instruction scheduling, they simply assume that any memory operation finishes after n-cycles. When the executed operation takes longer than presumed, the entire processor is stalled. As a result, often a large difference exists between the expected and the effective performance of the processor. For instance, a modulo scheduler [12] generates a memory access schedule of the inner-loops of 460 cycles[1] (Fig. 2-a). However, the actual performance varies between 540 and 740 cycles. The schedule takes longer than expected because the processor has at least to serialize the parallel accesses to D in loop 31. Extra stalls occur depending on whether the linker has assigned the C,B and/or D to the same memory.

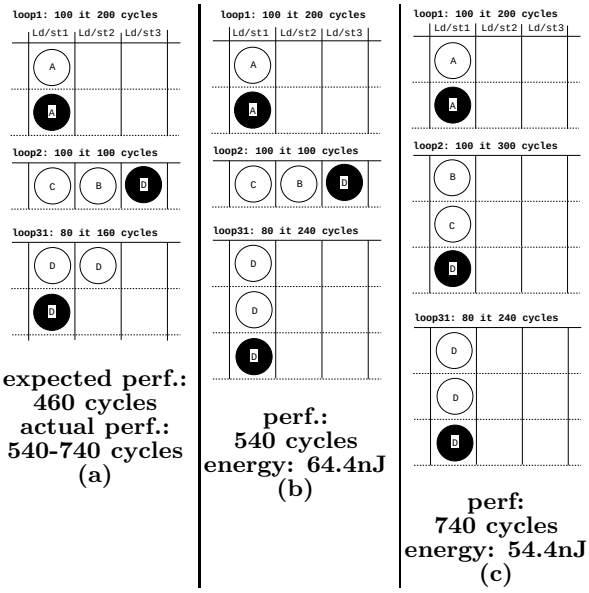


Figure 2: Empty issue slots in the memory access schedule of the inner-loops: (a) existing compiler; (b) with fastest partial data assignment; (c) with most energy efficient partial assignment

Because how the linker assigns the data to the memories has such a large impact on the performance of the system, [17] optimizes the data assignment and/or the memory schedule together. They impose restrictions on the assignment such that the energy is optimized, but still guarantee that the time-budget is met. The assignment constraints are modeled with a conflict graph (e.g., Fig. 3-left). The

[1]Note how the modulo scheduler schedules read/write operations from the same instruction in the same cycle.

nodes correspond to the data structures of the application. An edge between two data structures indicates that we need to store the data in different memories. Hence, the corresponding accesses to these data structures can be executed in parallel. For instance the edge between A and C forces us to store both data structures in different memories. The schedule for this conflict graph takes 540 cycles (Fig. 2-b). It consumes 64.4 nJ[2], because the conflict edges force us to store both C and B in a large memory (see complete assignment in Fig. 3-left).

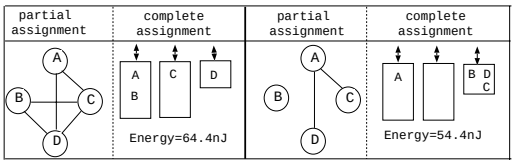


Figure 3: Partial assignment expressed with a conflict graphs: (left) fast; (right) more energy-efficient

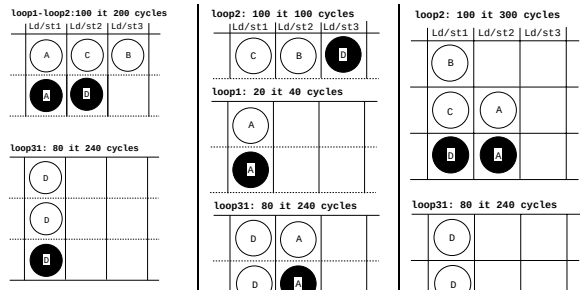
We can decrease its energy cost by reducing the number of conflicts (see Fig. 3-right). We can then more easily derive an energy efficient assignment. E.g., note how the small data structures B, D and C can now be assigned in the smallest most energy efficient memory. The energy consumption is then 54.4nJ instead of the original 64.4nJ. Less conflicts also implies that less memory accesses can execute in parallel. The code now takes 740 cycles (Fig. 2-c). The energy savings thus come at a performance loss.

However, many memory access slots remain empty. This is mainly because: (1) inter-iteration dependencies. For instance the minimum initiation interval of loop 1 is two, because A depends on itself. As a result, only 30% of the available memory slots is used; (2) we do not use power hungry multi-port memories. Hence, we cannot schedule operations that access the same data in parallel. E.g. in loop 31 we cannot execute the accesses to D in parallel.

With loop fusion we can further reduce the execution time of the code. There are two different ways to compact the application's schedule. On the one hand, we can fuse loops 1 and loop 2 for the fastest conflict graph (Fig. 3-left). The resulting schedule takes 440 cycles (Fig. 4-a). On the other hand, we can fuse loop 1 and loop 31 (Fig. 4-b). The schedule takes then only 380 cycles compared to 540 cycles for the non-fused code. We automatically generate the fused code for this decision (Fig. 1-right). In both cases, the energy cost remains the same because we keep the same conflict graph. If we change the conflict graph, we need to take different fusion decisions. E.g., for the more energy efficient conflict graph (Fig. 4-c), it is more beneficial to fuse loop 1 and loop 2. The execution time is then 540 cycles compared to 740 cycles for the non-fused code.

The fusion decisions and consequently, the performance of the application, thus heavily depend on the conflict graph. The more conflicts the higher the application's performance, but the more energy hungry it becomes. The problem is *to detect the most energy-efficient conflict graph and fusion decisions such that we can just meet the application's time-budget*. We outline our approach to address this problem in the next section.

[2]We compute the energy consumption as follows: $\sum_{\forall m \in M} \sum_{\forall ds \in m} \mathrm{NrAccess(ds)} \mathrm{E}^{m}_{\mathrm{access}}$



perf. 440 cycles energy: 64.4nJ (a)

perf. 380 cycles energy: 64.4nJ (b)

perf. 540 cycles energy: 54.4nJ (c)

Figure 4: Loop fusion fills the issue-slots: (a)-(b) two possibilities for the fastest partial data assignment (c) best fusion for the energy-efficient partial assignment

4. CONSTRUCTING AN ENERGY/ PERFORMANCE TRADE-OFF

We follow a constructive approach to solve the above problem (Algo. 1). We generate several assignment/fusion solutions for a range of time-budgets (line 11). Each solution is optimized for energy. It consists of the fused code together with data assignment decisions needed to obtain the predicted time-budget. The designer can then simply select the solution which best fits his needs.

After generating our internal model from the input C-code (line 3: $\mathcal{L}$), we build the assignment which results in the fastest possible solution (line 4). In this assignment, accesses to different data can be scheduled in parallel. However, we exclude parallel accesses to the same data structure, since this would require energy-hungry multi-port memories. This assignment corresponds to a conflict graph in which edges exist between each pair of data, but no self-edges. We then fuse as many loops as possible and schedule as many memory operations as possible in parallel. We call *greedy-loopfusion* for this purpose (line 6). This function greedily fuses as many loop pairs as beneficial for performance. It takes the assignment constraints into account while deciding which loops to fuse. We explain this function more in detail in the next section. It returns the transformed loop nests ($\mathcal{L}'$), the resulting execution time and annotates the conflict graph (see below). Thereafter, we quantify the energy cost of the fusion decisions (line 7). Basically, we assign the application's data to the memories of our architecture while optimizing the energy cost. The assignment takes all conflict constraints into account. We have implemented for this purpose an integer linear program, but other optimization heuristics can be used as well [14]. If the generated solution proves to be optimal for energy/performance, we store it and dump its C-code after fusion (line 8).

During each following iteration of the algorithm (line 6-10), we remove the conflict edge that has the smallest impact on the performance, but increases the assignment freedom the most (line 9). Hence, we can incrementally create more

energy-efficient data assignments, while keeping the performance loss in each step limited. For this purpose, we annotate each edge with the edge weight (e_{weight}). It quantifies each edge with the number of times that the compiler schedules accesses to the corresponding data structures in parallel. This number is a side-product of the *greedyloopfusion* function. The higher the edge weight, the more important this edge is for the performance of the application. We remove the edge which has the highest impact on the assignment freedom because it affects most accesses (line 9: numerator), but the smallest impact on the performance (line 9: denominator). The edge weight heavily depends on which operations are executed in parallel and thus on which loops are fused. Whenever we rerun *greedyloopfusion*, different loops may be fused and the edge weights can change considerably. Therefore, to correctly steer the order in which we remove edges, we update the weights after each fusion decision(line 6.3).

Algorithm 1 Generating performance/energy trade-off

1: **Input:** Initial C-code
2: **Algorithm:**
3: Parse C-code and build a list of loop nests: $\mathcal{L} = l_1..l_k$
4: Generate fastest possible assignment, i.e. a conflict graph CFG(DS, E) with edges between each pair of different data structures
5: **repeat**
6: $\mathcal{L}', \text{CFG}', \text{time} = \text{GreedyLoopFusion}(\mathcal{L}, CFG)$ where
   1. the fused loops $\mathcal{L}'$
   2. an estimation of the execution time: time
   3. annotate $e \in$ E with e_{weight}, the number of times the corresponding data are accessed together.
7: Evaluate energy cost for $\mathcal{L}'$
8: If Pareto-optimal, store <CFG, cost, time>-tuple and generate C code from $\mathcal{L}'$, the set of transformed loops
9: Remove $e(ds1, ds2) \in$ E with highest $(\text{NrAccess}(ds1) + \text{Nraccess}(ds2))/e_{\text{weight}}{}^3$
10: **until** $E = \emptyset$
11: **Output:** Pareto-set of (code, CFG, cost, time)-solutions

5. GREEDY LOOP FUSION

In applications, usually more than one pair of loops nests which can be fused. To maximize the performance, we therefore fuse loops until no pair remains which reduces the execution time after fusion. We explain here a heuristic to decide which loops to combine first. The input of our algorithm is an initial description of the loops, their statements and iteration domains. We also take the data assignment constraints into account, because they have a large impact on the fusion decisions (see example in Sect. 3).

In our algorithm, we first enumerate all possible fusion candidates (line 3). We represent them with a Fusion Graph (FG) (similar to [5]). The nodes of the graph correspond to the loops in the code. An edge between a pair of loops marks fusible loops. We conservatively assume that loops are only fusible when no (direct/or indirect) dependencies exist between them (line 3.1). Furthermore, when fusing two loops, the number of data structures which needs to be stored on the local memories increases. To ensure that a data assignment remains possible after fusion, in each iteration the required memory size should not exceed the size

Algorithm 2 GreedyLoopFusion

1: **Input:**
   1. fusion candidates: $\mathcal{L} = l_1..l_k$
   2. assignment constraints: $CFG(DS, E)$
   3. architecture description: memory latency, number of ports, available size
2: **Algorithm:**
3: Build fusion graph FG($\mathcal{L}$, E):
   An edge exists between two loops l1,l2 if:
   1. no dependences exists between l1 and l2
   2. the required memory size by both loops does not exceed the available size on the architecture.
4: Annotate each edge with fusion gain, i.e.
   1. Compute the initial performance: $t_{\mathcal{L}} = \sum_{\forall l_k \in \mathcal{L}} i_k * \text{MinII}(l_k, \text{CFG})$
   2. Compute for each edge l1-l2: $\text{FusionGain}(l1, l2) = (\text{MinII}(l1, \text{CFG}) + \text{MinII}(l2, \text{CFG}) - \text{MinII}(l1l2, \text{CFG})) * i12)/t_{\mathcal{L}}$

   where $i12$ is the number of iterations from both loops we can fuse. where MinII(l,CFG) is the minimum initiation interval of l_k. This schedule is constrained to the partial assignment, CFG.
5: **while** $\exists l1, l2 \in \mathcal{L}; FusionGain(l1, l2) > \alpha$ **do**
6: Select the loop pair (l1,l2) with the highest fusion gain
7: Fuse(l1,l2) and update FG
8: **end while**
9: Update the edge weights of CFG:
   compute how many times the corresponding data are accessed in parallel $\mathcal{L}'$ from the schedule generated by $\text{MinII}(l_i, \text{CFG})$
10: **Output:**
   1. fused code and its performance: $\mathcal{L}'$ and $t_{\mathcal{L}'}$
   2. updated conflict graph: $CFG'(D, E')$

provided by the architecture (line 3.2).

The edges are annotated with the *fusion gain* (line 4). The fusion gain is an estimation of the relative system's performance gains after fusion. We first estimate the initial application performance ($t_{\mathcal{L}}$). It is the sum of the schedule length of each basic block in the code (line 4.1). We estimate the schedule length of a basic block with an iterative modulo scheduler [12]. It is the product of its minimum initiation interval (MinII) with its number of iterations (i_k). Our modulo scheduler takes the number of memory ports and the access latency into account. We only schedule the memory operations and thus omit all other instructions. This simplifies the performance estimation, but remains sufficiently accurate because we focus on data dominated applications. We also take the assignment constraints into account during scheduling. Particularly, we only schedule memory operations in parallel if a conflict exists between their corresponding data in the conflict graph. After computing the initial performance, we compute for each pair of fusible loops the performance gains after fusion (4.2). It is the product of the fusion gain per iteration times the number of fused iterations. The fusion gain per iteration is the difference of the initiation interval of the fused length with the sum of those of the original loops. Our fusion tool can also estimate i_{12}, how many iterations of two loop nests can be overlapped [7].

After computing the fusion gains, we iteratively fuse the loop pair with the highest fusion gain (lines 5-8). We impose that the fusion gain should be larger than a threshold α

application	nr. loop nests	max. depth	perfectly nested
mm	2	3	no
dct/idct	1	4	no
rgb2yuv	2	3	no
fir	1	3	no
wave	3	2	no
conv	1	4	no
cmp	1	2	yes

Table 1: The benchmark applications: number of loops inside each application(2), maximum depth of a loop nest(3), (im)perfectly nested(4)

(line 5). In this way, we prevent fusing loops which have no impact on the performance, but would only generate extra control overhead. We use loop morphing to fuse the loops [7]. Despite this technique also combines loops with incompatible headers, the same limitations apply as for most loop transformation techniques which are based on the polyhedral model (see [15] for a description of the limitations). After every fusion step, we also re-evaluate which loop pairs can be combined and re-compute the fusion gains for the newly generated loops.

As long as profitable loop pairs remain, we repeat lines 5-8. Thereafter, we generate information to decide which conflict edges to remove first. We update the edge weights of the conflict graph based on the newly generated loops and their memory accesses schedule (line 9). The complexity of our loop fusion algorithm is proportional with the number of times we call the MinII -$O(m*m)$ where m is the number of fusible loops.

6. EXPERIMENTS

We quantify our approach with several tasks extracted from multimedia applications (see Tab. 1). *mm* is a matrix multiplication. *fir* is a finite impulse response filter. *conv* convolves an image with a 3x3 convolution kernel. *dct* and *wave* respectively stand for the DCT and wavelet transformation. *cmp* compares two images, writing the maximum value of each pixel in a third matrix. Finally, *rgb2yuv* implements a YUV to RGB transformation of an image. Real systems (e.g., MPEG4 IM1 player and MPEG21 3D encoder) typically execute several of these image/video processing tasks. To mimic their behavior, we combine these media applications in typical tasks-sets or scenarios.

We measure the execution time of the fused code with the Trimaran VLIW environment. We have modified its compiler to make it aware of the data assignment constraints. Only if an edge exists between two data structures in a conflict graph, the compiler schedules the corresponding memory accesses in parallel. All other accesses are sequentially scheduled. We estimate the energy cost of each fusion solution after assigning the data to the available memories [14]. As an example, we model a memory architecture which consists of four single port memories a 1kB-0.05 nJ/access, a 4kB-0.103 nJ/access, a 8kB-0.12 nJ/access and a 32kB-0.229 nJ/access (based on Cacti 3.0). Each memory has a two cycle access latency (configured in Trimaran). Obviously, other memory architectures can also be configured.

We first illustrate how our fusion approach generates a performance/energy trade-off. In Fig. 5 we automatically generate a trade-off for *fir_wave_conv*. Point 1 corresponds to the fastest solution. We obtain it after applying *greedy-loopfusion* on the original code and using the fastest assignment. Its conflict graph thus contains many edges which make it difficult to find an energy-efficient assignment. From this starting point, our algorithm iteratively reduces the energy cost by removing conflicts. Then, it clusters more data structures in energy efficient memories, but this increases the execution time (see points 2-5). Since in each point different assignment constraints are imposed, we also take other fusion decisions. E.g., in point 1 we fuse four loops whereas in point 6 we do not fuse any loop. No edges exist in the conflict graph of point 6, Hence, no parallel accesses are allowed. Fusion to enable more parallel accesses is then not beneficial.

As a reference (original curve), we indicate the trade-off by optimizing the data assignment/instruction scheduling, but without fusion [17]. With fusion we obtain a similar performance for a more energy efficient assignment. E.g., point 4 improves the performance of the fastest non-fused point (point 1nf) with 12% and is still 5% more energy efficient. Obviously, we can also improve the performance (e.g., compare points 1 to 1nf). Finally, both curves overlap at their end points, where fusion is not beneficial (6 and 3nf).

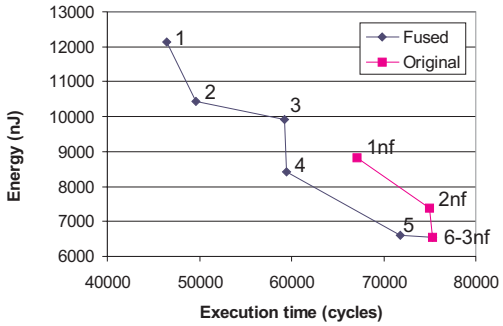


Figure 5: Existing techniques vs. our fusion-based approach for *fir_wave_conv*

We can generate similar trade-offs for other benchmarks (Fig. 6). The bars indicate the performance whereas the curves indicate the energy cost. The leftmost point of each benchmark represents the fastest and most energy consuming point. Moving to the right, we reduce the energy consumption at the expense of performance. The range of the trade-off depends however strongly on the application. E.g., *fir_wave_conv* offers a 40% performance improvement. In contrast, in *mm_fir* we can only improve the performance by 25%. Two reasons exist for this. First, the access schedule of the loops are more empty in some tasks than for others. For instance, the loops of *yuv* and *wave* often perform accesses to the same array. The compiler schedules these accesses sequentially, which results in longer and more empty access schedules. When fusing these loops with other ones, we can recuperate the empty access slots and find faster access schedules. This explains the large performance improvement of *fir_wave_conv*: the execution of *fir* and *conv* may be hidden in the loops of *wave*. Secondly, how well our technique fuses loops also depends on the initial loop headers. The more compatible they originally are, the less extra control overhead we need when fusing them. Furthermore, when the lengths of the loops are too unbalanced, the performance improvement is limited. In spite of these variations, we improve the performance on average with 27% compared to the fastest schedule obtained by [17].

The more loops we fuse, the more data structures coexist during the execution of the loop body and thus the more

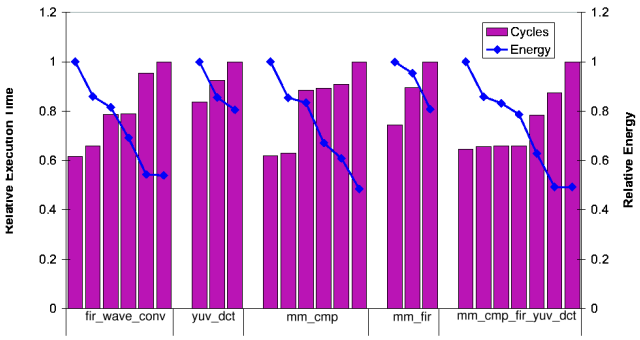


Figure 6: Energy vs. performance trade-off

memory is required. Hence, how aggressively we can fuse depends on the available memory size. We quantify the influence of memory size on the maximal attainable performance in Fig. 7. To obtain the performance bound, we assume that the memory layer consists of a single 4-port memory[4]. We apply our loop fusion algorithm for different memory sizes. We always use the fastest possible assignment. In this example, every time the memory size is increased with 1024 bytes, the execution time reduces with 10%. Hence, the performance gains attainable with fusion are limited by the available memory size.

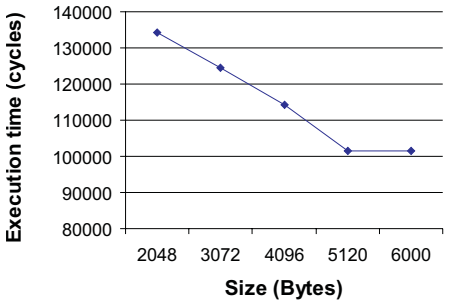


Figure 7: Influence of memory size on the maximal attainable performance for *mm_fir_wave_cmp*

An alternative way exists to exploit fusion to reduce the energy cost. With fusion we can use for the same performance an architecture with slower memories. Since slower memories consume less energy per access, this allows us to reduce the energy cost. We quantify the potential energy savings in Tab. 2. For this experiment, we use a VLIW architecture with four memory ports fully connected to four 8kB memories. We measure the execution time of *mm_fir_wave_conv* for the original non-fused and fastest fused code. We repeat the experiment for an architecture on which the memories have respectively two, three and four nano seconds access-time. Under the assumption that the processor runs at 1Ghz, this corresponds to a two, three and four cycles latency for each memory access. The last column indicates the energy consumption for each version. The memory energy models were borrowed from a major vendor for memories with two, three and four nano seconds access-time. Since we use a homogeneous memory architecture, the energy cost of the fused and non-fused code are the same for each access latency.

[4]This architecture is very power-hungry and not realistic for low-power systems. However, we only use it to quantify the upper bound of the performance.

Access Latency (cycles)	Execution Time (cycles)		Energy (nJ)
	Fused	Non Fused	
2	123715	**161665**	23379
3	**135675**	185986	17051
4	172285	209507	14156

Table 2: Fusion enables more energy-efficient memory architectures

The experiments show that the 3-cycles latency/fused version executes faster than the 2-cyles latency/non-fused one. At the same time, its energy cost is 30% cheaper, because increasing the access latency from two to three cycles reduces the energy per access. Similar results exist when increasing the access latency from three to four cycles. With fusion we can thus obtain the same performance with a cheaper memory architecture. Our tool allows a designer to explore which memory latency best fits his needs.

7. CONCLUSIONS

In this paper, we have discussed a loop fusion technique to globally optimize the memory bandwidth. Our tool decides which loops to fuse, while taking memory size, number of ports, access latency and feasible data assignments into account. Experimental results indicate that our approach outperforms existing techniques both in energy and performance. Despite our tool even fuses loops with incompatible headers, fusion remains limited to regular code [15]. We are currently removing this limitation to expand the applicability of our technique.

8. ADDITIONAL AUTHORS

Additional authors and acknowledgements: Sven Verdoorlaege (Kuleuven, Leuven, email: skimo@kotnet.org), Luis Piuel (DACYA U.C.M., Madrid, email: lpinuel@dacya.ucm.es). The authors also acknowledge the IWT Flanders and the Spanish Grant TIC 2002-0750

9. REFERENCES

[1] O. Avissar, R. Barua, and D. Stewart. Heterogeneous Memory Management for Embedded Systems. In *Proc. Cases*, 2001.

[2] F. Bodin, W. Jalby, C. Eisenbeis, and D. Windheiser. A quantitative algorithm for data locality optimization. In *Proc. Int. Wkshp. on Code Generation*, pages 119–145, 1991.

[3] D. Gannon and W. Jalby abd K. Gallivan. Strategies for cache and local memory management by global progra, optimizations. *J. of Parallel and Distributed Systems*, 25:587–617, 1988.

[4] P. Grun, N. Dutt, and A. Nicolau. Memory Aware Compilation through Timing Extraction. In *Proc. 37th Dac*, pages 316–321, Jun. 2001.

[5] K.McKinley, S. Carr, and C.Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[6] L. Lamport. The parallel execution of do-loops. *Communications of ACM*, 17(2):83–93, Feb. 1974.

[7] P. Marchal, J.I. Gomez, and F. Catthoor. Loop morphing to improve the performance on a VLIW. In *accepted for ASAP 2004*, 2004.

[8] M.Wolf. Improving locality and parallelism in nested loops. Technical report, Technical report CSL-TR-92-538, Stanford Univ., CA, USA, Sep. 1992.

[9] P. Panda, N. Dutt, and A. Nicolau. Exploiting Off-Chip Memory Access Modes in High-Level Synthesis. In *Proc. Iccad*, pages 333–340, Oct. 1997.

[10] P.Panda, F.Catthoor, N.Dutt, K.Danckaert, E.Brockmeyer, C.Kulkarni, A.Vandecappelle, and P.G.Kjeldsberg. Data and Memory Optimizations for Embedded Systems. *ACM Trans. on Design Automation for Embedded Systems (TODAES)*, 6(2):142–206, Apr. 2001.

[11] Y. Qian, S. Carr, and P. Sweany. Loop Fusion for Clustered VLIW Architectures. In *Proc. Joint Conference on Languages, Compilers and Tools for Embedded Systems and Software and Compilers for Embedded Systems*, pages 19–21, June 2002.

[12] B. Rau. Iterative Modulo Scheduling. Technical report, HP Labs, 1995.

[13] M. Saghir, P. Chow, and C. Lee. Exploiting Dual Data Banks in Digital Signal Processors. In *ASPLOS*, Jun. 1997.

[14] A. Vandecappelle, M. Miranda, E. Brockmeyer, F. Catthoor, and D. Verkest. Global Multimedia System Design Exploration using Accurate Memory Organization Feedback. In *Proc. 39th DAC*, 1999.

[15] S. Verdoorlaege, M. Bruynooghe, G. Janssens, and F. Catthoor. Multi-dimensional incremental loop fusion for data locality. In *Proceedings 2003 Application-specific Systems, Architectures and Processors*, pages 17–27, 2003.

[16] W. Verhaegh, E. Aarts, P. van Gorp, and P. Lippens. A Two-stage Solution Approach for Multidimensional Periodic Scheduling. *IEEE Trans. Computer Aided Design of Integrated Circuits and Systems*, 10(10):1185–1199, Oct. 2001.

[17] S. Wuytack, F. Catthoor, G. De Jong, and H. De Man. Minimizing the required memory bandwidth in VLSI system realizations. *IEEE Trans. VLSI Systems*, 7(4):433–441, Dec. 1999.