

Optimizing Local Memory Allocation and Assignment Through a Decoupled Approach

Boubacar Diouf, Özcan Özturk, Albert Cohen

▶ **To cite this version:**

Boubacar Diouf, Özcan Özturk, Albert Cohen. Optimizing Local Memory Allocation and Assignment Through a Decoupled Approach. The 23rd International Workshop on Languages and Compilers for Parallel Computing, Oct 2009, Newark, Delaware, United States. hal-00645302

HAL Id: hal-00645302
https://inria.hal.science/hal-00645302

Submitted on 27 Nov 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Optimizing Local Memory Allocation and Assignment Through a Decoupled Approach

Boubacar Diouf,[1] Ozcan Ozturk,[2] Albert Cohen [1]

[1] INRIA Saclay and Paris-Sud 11 University, [2] Bilkent University

Abstract. Software-controlled local memories (LMs) are widely used to provide fast, scalable, power efficient and predictable access to critical data. While many studies addressed LM management, keeping hot data in the LM continues to cause major headache. This paper revisits LM management of arrays in light of recent progresses in register allocation, supporting multiple live-range splitting schemes through a generic integer linear program. These schemes differ in the grain of decision points. The model can also be extended to address fragmentation, assigning live ranges to precise offsets. We show that the links between LM management and register allocation have been underexploited, leaving much fundamental questions open and effective applications to be explored.

1 Introduction

Software-controlled local memories (LMs) are widely used to provide fast, predictable, and power efficient access to critical data. Compilation-time LM management has been connected to register allocation for at least 30 years [1]. Register allocation is just emerging from a series of tremendous, fundamental and applied progresses [2–6], redefining design of compiler backends. Surprisingly, this revolution was completely missed in the domain of LM management.

In the following, *main memory* (MM) refers to the main — typically off-chip DRAM — memory resources; *local memory* (LM) refers to — typically on-chip SRAM — low-latency, high-bandwidth memories to exploit temporal locality and hide the cost of accessing the MM. We will only consider *single-threaded* code running on a *single LM*. Extending to multiple LMs with different characteristics should be analogous to register allocation on multiple register classes and register files; but multithreaded LM management will be more challenging.

Most ARM processors have an on-chip LM [7], and more generally, it is typical for DSPs and embedded processors to have LMs, also called scratchpad memories [8, 9]. More specialized processors also utilize LMs, including stream-processing architectures such as graphical processors (GPUs) and network processors [10, 11]. Most processor(s) may directly access the MM, but few exceptions exist. The IBM Cell broadband engine's synergistic processing units (SPU) [12] which rely exclusively on DMA for instruction and data transfers with MM. Our approach is compatible with such memory models.

The outline of the paper is the following. Section 2 sets the landscape for LM management into the state-of-the-art of decoupled and SSA-based register

allocation. Section 3 discusses related work. Section 4 formalizes the optimization problem and details a generic integer linear programming solution. Section 5 validate our approach and model on simple yet representative benchmarks.

2 Motivation

Recent progress in register allocation leverage the complexity and optimality benefits of decoupling its allocation and assignment phases [2, 5]. The allocation phase decides which variables to spill and which to assign to registers. The assignment phase chooses which variable to assign to which register.

The allocation phase relies on the maximal *number* of simultaneously living variables, called MaxLive, a measure of *register pressure*. If at some program point the pressure exceeds the number of available registers, MaxLive needs to be reduced through spilling. When "*fine-grain enough*" live-range splitting is allowed, it is sufficient that MaxLive is less or equal to the number of available registers for all the live ranges to be allocated, and to guarantee that the forthcoming assignment phase can be done without further spilling. In many cases, optimal assignment can be achieved in linear time [5]. This decoupled approach permits to focus on the hard problem, namely the spilling decisions. It also improves the understanding of the interplay between live-range splitting and the expressiveness and complexity of register allocation. This is best illustrated by the success of SSA-based allocation [3–5, 13].

The intuition for decoupled register allocation derives from the observation that live-range splitting is almost always profitable if it allows to reduce the number of register spills, even at the cost of extra register moves. The decoupled approach focuses on spill minimization only, pushing the minimization of register moves to a later register coalescing phase [2, 14]. Here again, SSA-based techniques have won the game, collapsing the register coalescing with the hard problem of getting out of SSA [4, 15], as one of the last backend compiler passes.

The domain of LM management tells a very different story. Some heuristics exist [16–19] but little is known about the optimization problem, its complexity and the interplay with other optimizations. The burning hot question is of course: does the decoupled approach hold for the LM management problem? Surprisingly, the state-of-the-art of LM management completely ignores all the advances in register allocation. When focusing on arrays, the similarity between register and LM allocation is obvious nonetheless:

LM allocation. Deciding which array blocks to spill to MM and which array blocks to allocate to the LM. Spilling is typically supported by DMA units.

LM assignment. Deciding at which LM offset to assign which allocated array block. There is no unique equivalent of register moves when reconciling offsets across control-flow regions: most papers assume a cheap local copy operation (compared to spilling to MM); one may also address each LM-allocated array block with its own dedicated pointer, bringing the cost down to a plain pointer copy (at the expense of register pressure).

The motivation for decoupled LM management is all the more appealing that LM offset reconciliation can be implemented with pointer copies. With such an assumption, the array move/spill ratio becomes negligible w.r.t. the already low register move/spill ratio. To the best of our knowledge, this paper is the first to explore a decoupled approach to LM management.

Figure 1 shows a code fragment from the UTDSP benchmarks [20]. It has been slightly simplified for the sake of the exposition, preserving the original array data flow. Since arrays are frequently accessed inside loops, LM management algorithms often (re)consider allocation decisions at loop entry points; a form of *live-range splitting*. On this code fragment, this would make accesses to `C` to artificially coincide with accesses to `F`. Zooming to the grain of individual program statements, the live ranges of these two arrays are in fact disjoint. Finer grain decision points would be beneficial. It is interesting to take allocation decisions for each array at the points where it is going to be frequently accessed (if its usage compensate memory transfers). Of course, the cost of (un)loading whole array blocks to the LM is too high to authorize live-range splitting at every instruction (a.k.a. load/store optimization [2, 6]). More advanced strategies would integrate loop transformations such as loop distribution and strip-mining [21].

3 Related Work

Strong links between register allocation and local memory management have been discovered for more than 30 years by Fabri [1]. This seminal paper also studied the interplay with loop transformations. It has been ignored in the field of LM management, as well as 2003-2009 register allocation [2, 4–6].

Previous studies addressed LM management from different angles, targeting both code and data. We are interested in data management [18, 22, 23], and target dynamic methods which are superior to static ones except when code size is extremely constrained [16]. We elaborate on two recent series of results targeting stack and global array management in LMs, embracing the analogies with register allocation. The first approach [19] uses an existing graph coloring technique to perform memory allocation for arrays. It paritions the LM for each different array size, performs live range spliting and use a register allocation framework to perform memory coloring. The second approach [16, 17, 24] allocates data onto the scratch-pad memory between program regions separated by specific program points. More specifically, allocation is based on the access frequency-per-byte of a variable in a region (collected from profile data). Program points are located at the beginning of a called procedure or before loop entry.

Our work leverages the decoupled allocation/assignment approach, allowing for scalable and more effective algorithms. It also offers much more flexibility in terms of integration of architecture constraints and performance models.

4 Decoupled Local Memory Management

The previous example shows the impact of the decision points for live-range splitting, to take advantage of the dynamic behavior of the program. Having

```
// Nested within outer loops
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    C[i][j] = /* ... */;

F[0][0]=1; F[0][1]=2; F[0][2]=1;
F[1][0]=2; F[1][1]=4; F[1][2]=2;
F[2][0]=1; F[2][1]=2; F[2][2]=1;
```

Fig. 1. Example: `Edge-Detect`

```
for (i=0; i<N; i++)
  // Outer strip-mined loop
  for (jj=0; jj<N+B-1; jj+=s)
    // Inner strip-mined loop
    for (j=jj; j<N && j<jj+s; j++)
      C[i][j] = /* ... */;

F[0][0]=1; /* ... */ F[2][2]=1;
```

Fig. 2. Homogeneous blocks

```
for (i=0; i<N; i++)
  // Outer strip-mined loop
  for (jj=0; jj<N+B-1; jj+=s)
    STORE(C[i][jj..min(jj+B-1,N-1)]);

STORE(F[0..2][0..2]);
```

Fig. 3. Abstract model

set the context of the optimization problem, we formalize the optimization as a generic integer linear program (ILP).

4.1 Preliminary Analyses and Transformations

We make several assumptions. We currently restrict our approach to uniform array accesses only. Non-uniform accesses can be exploited through approximations or pre-pass loop and data-layout transformations [18, 25, 26]. We consider loop nests that have been previously tiled for locality, together with data layout compaction and uniformization. As a rule of thumb, those transformations should favor the emergence of "homogeneous" array blocks, meaning that for a given array `A`, the loop transformations must strive to strip-mine the computation such that inner loops traverse a fixed-size block of `A` in a dense manner. Most numerical and signal-processing codes exhibit a vast majority of dense, uniform access patterns. In such cases, homogenization is trivial: it suffices to set the same strip-mining factor for all loops operating over a given array [17]. The last step is to abstract the transformed code, isolating array block operations into atomic regions. Considering the motivating example, these two steps are illustrated on Figures 2 and 3; s is the homogeneous block size for array `C`.

This abstraction is sufficient to collect profile information on the homogenized version of the loop nest, then to solve the allocation and assignment problems. When generating the code for a real target, one needs to insert code for array block load/store, and pointer moves. Again, standard techniques exist when array accesses are uniform [17, 18].

On the abstracted program, array blocks play the role of scalar variables in a register allocation problem, with the exception of cost modeling. To benefit from its algorithmic properties, we extend the SSA form to operate on array blocks. This extension differs from Array SSA proposals [27, 28]: it does not attempt to model the data flow of individual array elements. In this form, array blocks are fully renamed, and name conflicts at control-flow points are handled with Φ functions following the rules of strict SSA form.

From array blocks, one need to extract *live ranges* which will be the subject of the allocation and assignment decisions. Live ranges generalize live intervals in basic blocks to arbitrary control flow. In SSA form, extraction of live ranges can be done in linear time [4]. Live ranges and detailed profiling of the application is used to generate frequency information.

Note that, moving array blocks within the LM at every decision point may add severe overhead, limitting the effective benefits of live-range splitting. To

avoid such penalties, we only move the base pointers to these blocks rather than moving the chunk of data. In fact, this low-cost solution may seem so appealing to the reader that she may wonder why it was not considered in state-of-the-art techniques. The main reason is of course that a decoupled allocation/assignment scheme is necessary to make sure that *all array block moves* induced by the assignment phase are associated with *offset reconciliation across different control-flow paths*. When resorting to a unified allocation-and-assignment algorithm with live-range splitting [17], there is no such guarantee. Some moves are associated with real needs for array block displacement. Another reason is the increased pressure on the registers: we consider that a register spill is a lot cheaper than LM block-copying, and even if repeated register spills occur in an inner loop (which is very unlikely on RISC or VLIW processors), it will not be more expensive than an array block displacement; further experimental analysis and tradeoffs should be explored in the future.

Following Udayakumaran et al. [16], we extend every basic block with a final program point. This extra point will be used to capture any live range load or eviction at the end of the basic block, isolating the formalization of this decision from the reconciliation of the decisions associated with incoming/outgoing control-flow arcs. Thanks to this extra point, the only cross-basic-block equations will be reconciliation equations.

4.2 Management Schemes

Once preliminary transformations have been applied, One must choose decision points. Live-range splitting can be implemented at different decision points depending on a customizable grain/aggressiveness. This tradeoff can be instantiated into three typical schemes:

Scheme 1. One could make decisions at fine granularity points, where a point indicates an — abstract array block — instruction. This will exploit the array allocation and assignment at instruction granularity, providing a full latency minimization. However, modelling this scheme as an ILP may incur excessive complexity, due to the number of decision points.

Scheme 2. The second approach is similar to the SSA-based register allocation approach, where one may only take allocation decisions at points where a live range becomes alive.

Scheme 3. The third approach is to make an allocation decision per array block, without any splitting. This approach is called *static* in the context of LM management, and corresponds to the spill-everywhere register allocation problem. In this case, there are no program/decision points, leading to a much simpler optimization problem. However, the connection between MaxLive and colorability is lost, and the execution latency will be higher compared to the previous schemes.

Allocation. As mentioned earlier, variables in the ILP correspond to live-range splitting points where allocation decisions are taken and memory transfer instructions may be inserted. The result of the ILP model is for each decision

point and according to the previous decision point: (i) live ranges remaining in the LM, (ii) live ranges evicted from the LM, (iii) live ranges loaded to the LM, (iv) live ranges kept in the MM.

Assignment. Due to fragmentation, colorability is not sufficient to guarantee allocation of consecutive array blocks. In practice, fragmentation-avoidance heuristics perform very well, even on rather constrained problems [17]. Therefore, a pragmatic approach consists in ignoring fragmentation in the allocation phase, and leads to a greedy approach like Belady's furthest first algorithm [5, 29]. Of course, when implementing Scheme 1, one must be careful to always *assign the same offset* to a given (LM-allocated) live range *within a given basic block*. Since live-range splitting at SSA definition points is sufficient to guarantee colorability, this restriction does not induce spurious spills.

5 Experimental Results

Benchmark	Brief description	Suite	Data size	arrays /blocks
`Edge-Detect`	Edge detection in an image	[20]	196644	4/385
`D-FFT`	256-point complex FFT	[20]	2032	7/7
`Bmcm`	Water molecular dynamics	[30]	125240	10/310
`MxM`	Matrix multiplication	n.a.	120000	3/300

Fig. 4. Application codes.

Constant	Latency
$latency_LM$	8
$latency_MM$	128
$latency_move(s_v)$	$8 + 2s_v$
$latency_spill(s_v)$	$128 + 4s_v$
$latency_reload(s_v)$	$128 + 4s_v$

Fig. 5. Model parameters.

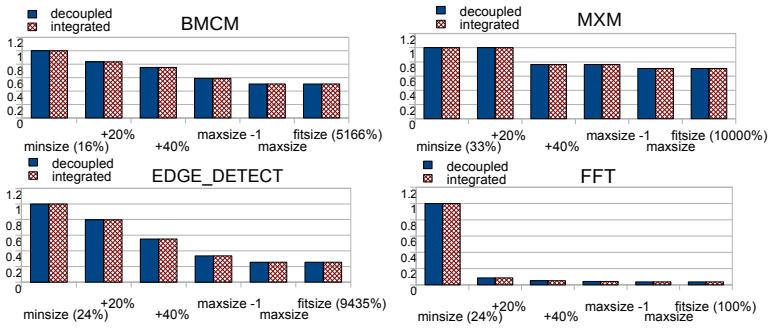


Fig. 6. Experimental results for decoupled and integrated approaches.

Experiments on real hardware are out of the scope of this work. We aim at validating the fundamental hypotheses and the relevance of the decoupled approach for LM management. We consider matrix multiplication and three array-intensive benchmarks from the UTDSP [20] and Perfect Club [30] suites. See Figure 4 and the model parameters in Figure 5. We use Scheme 2 for all experiments, a good tradeoff between expressiveness and complexity. Figure 6 shows the cost model results for each benchmark. The dark bar (left) refers to the cost

achieved by the allocation phase alone, selecting which live ranges will reside in the LM at which point. The light bar (right) refers to the cost achieved by integrated allocation-assignment, a very complex and inflexible optimization taking offsets and fragmentation into account. The second bar can only be identical or higher than the first (more constraints are taken into account). We consider different LM sizes: *minsize* corresponds to the the biggest array block in the program; *fitsize* is the total size of all array blocks; *maxsize* is the maximum of the total size of array blocks simultaneously alive at any given point; and the minimal size plus 20% (resp. 40%) of *maxsize*. We also state *minsize* and *fitsize* in percentage of *maxsize*. These results validate our two main insights:

1. The decoupled and integrated algorithms compute the same cost for all benchmarks and LM sizes. This confirms that decoupled management is as beneficial for LMs as it is for registers, in spite of the fragmentation issue.
2. For all benchmarks, the cost is optimal — same as for *fitsize* — as soon as the LM size reaches *maxsize*, while the cost immediately increases for smaller LM sizes (*maxsize-1*). This confirms that MaxLive is the relevant criterion.

6 Conclusion

The state-of-the-art in LM management ignores the tremendous advances in decoupled and SSA-based register allocation. We set up a new bridge between the two optimization problems. Our experiments validate the decoupling of the LM allocation and assignment stages: after an optimal allocation, we demonstrate a total absence of assignment-stage or fragmentation-induced spills.

Acknowledgements. This work was partly supported by the SARC FET FP6 and OMP IST FP7 european projects. We are also thankful to Fabrice Rastello and John Cavazos for tutoring us into the world of register allocation.

References

1. Fabri, J.: Automatic storage optimization. In: ACM Symp. on Compiler Construction. (1979) 83–91
2. Appel, A.W., George, L.: Optimal spilling for CISC machines with few registers. In: PLDI'01, Snowbird, Utah, USA (June 2001) 243–253
3. Bouchez, F., Darte, A., Guillon, C., Rastello, F.: Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? In: WDDD'06, Boston, MA (2006)
4. Hack, S., Grund, D., Goos, G.: Register allocation for programs in SSA-form. In: CC'06. (2006) 247–262
5. Bouchez, F., Darte, A., Guillon, C., Rastello, F.: Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? or revisiting register allocation: Why and how. In: LCPC'06. LNCS, New Orleans, Louisiana, Springer Verlag (2006)
6. Quintáo Pereira, F.M., Palsberg, J.: Register allocation by puzzle solving. SIGPLAN Not. **43**(6) (2008) 216–226

7. ARM: Document No. ARM DDI 0084D, ARM Ltd. ARM7TDMI-S data sheet (1998)
8. Motorola: M-CORE – MMC2001 reference manual, Motorola Corporation (1998)
9. Instruments, T.: TMS370Cx7x 8-bit microcontroller, Texas Instruments (1997)
10. NVIDIA: NVIDIA unified architecture GeForce 8800 GT (2008)
11. Burns, M., Prier, G., Mirkovic, J., Reiher, P.: Implementing address assurance in the Intel IXP (2003)
12. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the Cell multiprocessor. IBM Journal of Research and Development **49**(4/5) (2005)
13. Bouchez, F., Darte, A., Rastello, F.: On the complexity of register coalescing. In: CGO'07. (2007)
14. Bouchez, F., Darte, A., Rastello, F.: Advanced conservative and optimistic register coalescing. In: CASES'08. (2008) 147–156
15. Boissinot, B., Darte, A., de Dinechin, B.D., Guillon, C., Rastello, F.: Revisiting out-of-SSA translation for correctness, code quality and efficiency. In: CGO'09. (2009) 114–125
16. Udayakumaran, S., Barua, R.: Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In: CASES'03. (2003) 276–286
17. Udayakumaran, S., Dominguez, A., Barua, R.: Dynamic allocation for scratch-pad memory using compile-time decisions. ACM Trans. Embed. Comput. Syst. **5**(2) (2006) 472–511
18. Kandemir, M., Ramanujam, J., Irwin, J., Vijaykrishnan, N., Kadayif, I., Parikh, A.: Dynamic management of scratch-pad memory space. In: DAC'01. (2001) 690–695
19. Li, L., Gao, L., Xue, J.: Memory coloring: A compiler approach for scratchpad memory management. In: PACT'05. (2005) 329–338
20. Lee, C.G.: UTDSP benchmarks (1998)
21. Wolfe, M.J.: High Performance Compilers for Parallel Computing. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
22. Dominguez, A., Nguyen, N., Barua, R.K.: Recursive function data allocation to scratch-pad memory. In: CASES'07. (2007) 65–74
23. Issenin, I., Brockmeyer, E., Miranda, M., Dutt, N.: DRDU: A data reuse analysis technique for efficient scratch-pad memory management. ACM Trans. Des. Autom. Electron. Syst. **12**(2) (2007) 15
24. Avissar, O., Barua, R., Stewart, D.: An optimal memory allocation scheme for scratch-pad-based embedded systems. ACM Trans. Embed. Comput. Syst. **1**(1) (2002) 6–26
25. De La Luz, V., Kandemir, M.: Array regrouping and its use in compiling data-intensive embedded applications. IEEE Trans. Comput. **53**(1) (2004) 1–19
26. Kodukula, I., Ahmed, N., Pingali, K.: Data-centric multi-level blocking. In: PLDI'97, Las Vegas, Nevada (June 1997) 346–357
27. Knobe, K., Sarkar, V.: Array SSA form and its use in parallelization. In: ACM Conf. on Principles of Programming Languages (PoPL'08), San Diego, CA (January 1998) 107–120
28. Rus, S., He, G., Alias, C., Rauchwerger, L.: Region array SSA. In: PACT'06. (2006) 43–52
29. Belady, L.A.: A study of replacement algorithms for virtual storage computers. 9th Annual ACM-SIAM Symposium on Discrete Algorithms (1966)
30. et al, M.B.: The perfect club benchmarks: Effective performance evaluation of supercomputers. International Journal of Supercomputer Applications **3** (1988) 5–40